AIX Version 7.2

# *Kernel Extensions and Device Support Programming Concepts*

**Note**

Before using this information and the product it supports, read the information in “Notices” on page 427.

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

# Contents

# About this document

This document provides system programmers with complete information about kernel programming and the kernel environment for the AIX® operating system. Programmers can use this book to gain knowledge of device drivers, kernel services, debugging tools, and kernel subsystems. Topics include the kernel environment, system calls, the kernel debug program, system dump, and virtual file systems. Each subsystem is described in detail.

## How to Use This Document

This document provides two types of information: (1) an overview of the kernel programming environment and information a programmer needs to write kernel extensions, and (2) information about existing kernel subsystems.

## 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.

# Kernel Extensions and Device Support Programming Concepts

The term kernel extension applies to all routines added to the kernel, independent of their purpose.

Kernel extensions can be added at any time by a user with the appropriate privilege. Kernel extensions run in the same mode as the kernel. That is, when the 64–bit kernel is used, kernel extensions run in 64–bit mode. Therefore, AIX supports 64–bit kernel extensions only.

## Kernel Environment

The kernel is dynamically extendable and can be expanded by adding routines that belong to any of the functional classes.

A process executing in user mode can customize the kernel by using the **sysconfig** subroutine, if the process has appropriate privilege. In this way, a user-mode process can load, unload, initialize, or terminate kernel routines. Kernel configuration can also be altered by changing tunable system parameters.

Kernel extensions can also customize the kernel by using kernel services to load, unload, initialize, and terminate dynamically loaded kernel routines; to create and initialize kernel processes; and to define interrupt handlers.

**Note:** Private kernel routines (or kernel services) execute in a privileged protection domain and can affect the operation and integrity of the whole system. See “Kernel Protection Domain” on page 22 for more information.

**Related concepts**

Locking Kernel Services
The kernel services can be locked using any of the methods that are described in this overview.

Handling Signals While in a System Call
Signals can be generated asynchronously or synchronously with respect to the thread that receives the signal. An asynchronously generated signal is one that results from some action external to a thread. It is not directly related to the current instruction stream of that thread. Generally these are generated by other threads or by device drivers.

System Calls Available to Kernel Extensions
The system calls are grouped according to which subroutines call them.

## Understanding Kernel Extension Symbol Resolution

The concepts included in this section explain about the kernel extension symbol resolution.

### Exporting Kernel Services and System Calls

A kernel extension provides additional kernel services and system calls by specifying an export file when it is link-edited. An export file contains a list of symbols to be added to the kernel name space. In addition, symbols can be identified as system calls for 32-bit processes, 64-bit processes, or both.

In an export file, symbols are listed one per line. These system calls are available to both 32- and 64-bit processes. System calls are identified by using one of the **syscall32**, **syscall64** or **syscall3264** keywords after the symbol name. Use **syscall32** to make a system call available to 32-bit processes, **syscall64** to make a system call available to 64-bit processes, and **syscall3264** to make a system call available to both 32- and 64-bit processes. For more information about export files, see **ld** Command.

When a new kernel extension is loaded by the **sysconfig** or **kmod_load** subroutine, any symbols exported by the kernel extension are added to the kernel name space, and are available to all subsequently loaded

kernel extensions. Similarly, system calls exported by a kernel extension are available to all user programs or shared objects subsequently loaded.

## Using Kernel Services

The kernel provides a set of base kernel services to be used by kernel extensions. Kernel extensions can export new kernel services, which can then be used by subsequently loaded kernel extensions.

Base kernel services, which are described in the services documentation, are made available to a kernel extension by specifying the **/usr/lib/kernex.imp** import file during the link-edit of the extension.

**Note:** Link-editing of a kernel extension should always be performed by using the **ld** command. Do not use the compiler to create a kernel extension.

If a kernel extension depends on kernel services provided by other kernel extensions, an additional import file must be specified when link-editing. An import file lists additional kernel services, with each service listed on its own line. An import file must contain the line `#!/unix` before any services are listed. The same file can be used both as an import file and an export file. The `#!/unix` line is ignored when a file is used as an export file.

## Using System Calls with Kernel Extensions

A restricted set of system calls can be used by kernel extensions.

A kernel process can use a larger set of system calls than a user process in kernel mode. "System Calls Available to Kernel Extensions" on page 30 specifies which system calls can be used by either type of process. User-mode processes in kernel mode can only use system calls that have all parameters passed by value. Kernel routines running under user-mode processes cannot directly use a system call having parameters passed by reference.

The second restriction is imposed because, when they access a caller's data, system calls with parameters passed by reference access storage across a protection domain. The cross-domain memory services performing these cross-memory operations support kernel processes as if they, too, accessed storage across a protection domain. However, these services have no way to determine that the caller is in the same protection domain when the caller is a user-mode process in kernel mode. For more information on cross-domain memory services, see "Cross-Memory Kernel Services" on page 71.

> **Note:** System calls must not be used by kernel extensions executing in the interrupt handler environment.

System calls available to kernel extensions are listed in **/usr/lib/kernex.imp**, along with other kernel services.

### *Loading and Unloading Kernel Extensions*

Kernel extensions can be loaded and unloaded by calling the **sysconfig** function from user applications.

A kernel extension can load another kernel extension by using the **kmod_load** kernel service, and kernel extensions can be unloaded by using the **kmod_unload** kernel service.

*Loading Kernel Extensions*

Normally, kernel extensions that provide new system calls or kernel services only need to be loaded once. For these kernel extensions, loading should be performed by specifying SYS_SINGLELOAD when calling the **sysconfig** function, or LD_SINGLELOAD when calling the **kmod_load** function.

If the specified kernel extension is already loaded, a second copy is not loaded. Instead, a reference to the existing kernel extension is returned. The loader uses the specified pathname to determine whether a kernel extensions is already loaded. If multiple pathnames refer to the same kernel extension, multiple copies can be loaded into the kernel.

If a kernel extension can support multiple instances of itself (particularly its data), it can be loaded multiple times, by specifying SYS_KLOAD when calling the **sysconfig** function, or by not specifying LD_SINGLELOAD when calling the **kmod_load** function. Either of these operations loads a new copy of the kernel extension, even when one or more copies are already loaded. When this operation is used,

currently loaded routines bound to the old copy of the kernel extension continue to use the old copy. Subsequently loaded routines use the most recently loaded copy of the kernel extension.

*Unloading Kernel Extensions*
Kernel extensions can be unloaded. For each kernel extension, the loader maintains a use count and a load count. The use count indicates how many other object files have referenced some exported symbol provided by the kernel extension. The load count indicates how many explicit load requests have been made for each kernel extension.

When an explicit unload of a kernel extension is requested, the load count is decremented. If the load count and the use count are both equal to 0, the kernel extension is unloaded, and the memory used by the text and data of the kernel extension is freed.

If either the load count or use count is not equal to 0, the kernel extension is not unloaded. As processes exit or other kernel extensions are unloaded, the use counts for referenced kernel extensions are decremented. Even if the load and use counts become 0, the kernel extension may not be unloaded immediately. In this case, the kernel extension's exported symbols are still available for load-time binding unless another kernel extension is unloaded or the **slibclean** command is executed. At this time, the loader unloads all modules that have both load and use counts of 0.

## Using Private Routines

So far, symbol resolution for kernel extensions has been concerned with importing and exporting symbols *from* and *to* the kernel name space. Exported symbols are global in the kernel, and can be referenced by any subsequently loaded kernel extension.

Kernel extensions can also consist of several separately link-edited modules. This is particularly useful for device drivers, where a kernel extension contains the top (pageable) half of the driver and a dependent module contains the bottom (pinned) half of the driver. Using a dependent module also makes sense when several kernel extensions use common routines. In both cases, the symbols exported by the dependent modules are not added to the global kernel name space. Instead, these symbols are only available to the kernel extension being loaded.

When link-editing a kernel extension that depends on another module, an import file should be specified listing the symbols exported by the dependent module. Before any symbols are listed, the import file should contain one of the following lines:

```
#! path/file
```

or

```
#! path/file(member)
```

**Note:** This import file can also be used as an export file when building the dependent module.

Dependent modules can be found in an archive file. In this case, the member name must be specified in the `#!` line.

While a kernel extension is being loaded, any dependent modules are only loaded a single time. This allows modules to depend on each other in a complicated way, without causing multiple instances of a module to be loaded.

**Note:** The loader uses the pathname of a module to determine whether it has already been loaded. Another copy of the module can be loaded if different path names are used for the same module.

The symbols exported by dependent modules are not added to the kernel name space. These symbols can only be used by a kernel extension and its other dependent modules. If another kernel extension is loaded that uses the same dependent modules, these dependent modules will be loaded a second time.

## Using Libraries

The operating system provides libraries that can be used by kernel extensions.

**Related concepts**

System Calls

A system call is a routine that allows a user application to request actions that require special privileges. Adding system calls is one of several ways to extend the functions provided by the kernel.

### *libcsys Library*

The **libcsys.a** library contains a subset of subroutines found in the user-mode **libc.a** library that can be used by kernel extensions.

When using any of these routines, the header file **/usr/include/sys/libcsys.h** should be included to obtain function prototypes, instead of the application header files, such as **/usr/include/string.h** or **/usr/include/stdio.h**. The following routines are included in **libcsys.a**:

- **atoi**
- **bcmp**
- **bcopy**
- **bzero**
- **memccpy**
- **memchr**
- **memcmp**
- **memcpy**
- **memmove**
- **memset**
- **ovbcopy**
- **strcat**
- **strchr**
- **strcmp**
- **strcpy**
- **strcspn**
- **strlen**
- **strncat**
- **strncmp**
- **strncpy**
- **strpbrk**
- **strrchr**
- **strspn**
- **strstr**
- **strtok**

**Note:** In addition to these explicit subroutines, some additional functions are defined in **libcsys.a**. All kernel extensions should be linked with **libcsys.a** by specifying **-lcsys** at link-edit time. The library **libc.a** is intended for user-level code only. Do not link-edit kernel extensions with the **-lc** flag.

### *libsys Library*

The **libsys.a** library provides the following set of kernel services:

- **d_align**

- **d_roundup**
- **timeout**
- **timeoutcf**
- **untimeout**

When using these services, specify the **-lsys** flag at link-edit time.

### *User-provided Libraries*

To simplify the development of kernel extensions, you can choose to split a kernel extension into separately loadable modules. These modules can be used when linking kernel extensions in the same way that they are used when developing user-level applications and shared objects.

In particular, a kernel module can be created as a shared object by linking with the **-bM:SRE** flag. The shared object can then be used as an input file when linking a kernel extension. In addition, shared objects can be put into an archive file, and the archive file can be listed on the command line when linking a kernel extension. In both cases, the shared object will be loaded as a dependent module when the kernel extension is loaded.

## Understanding Execution Environments

A kernel extension runs in the *process environment* when invoked either by a user process in kernel mode or by a kernel process.

A kernel extension is executing in the *interrupt environment* when invoked as part of an interrupt handler.

A kernel extension can determine in which environment it is called to run by calling the **getpid** or **thread_self** kernel service. These services respectively return the process or thread identifier of the current process or thread , or a value of -1 if called in the interrupt environment. Some kernel services can be called in both environments, whereas others can only be called in the process environment.

**Note:** No floating-point functions can be used in the kernel.

### Process Environment

A routine runs in the process environment when it is called by a user-mode process or by a kernel process.

Routines running in the process environment are executed at an interrupt priority of INTBASE (the least favored priority). A kernel extension running in this environment can cause page faults by accessing pageable code or data. It can also be replaced by another process of equal or higher process priority.

A routine running in the process environment can sleep or be interrupted by routines executing in the interrupt environment. A kernel routine that runs on behalf of a user-mode process can only invoke system calls that have no parameters passed by reference. A kernel process, however, can use all system calls listed in the System Calls Available to Kernel Extensions if necessary.

### Interrupt Environment

A routine runs in the interrupt environment when called on behalf of an interrupt handler. A kernel routine executing in this environment cannot request data that has been paged out of memory and therefore cannot cause page faults by accessing pageable code or data. In addition, the kernel routine has a stack of limited size, is not subject to replacement by another process, and cannot perform any function that would cause it to sleep.

A routine in this environment is only interruptible either by interrupts that have priority more favored than the current priority or by exceptions. These routines cannot use system calls and can use only kernel services available in both the process and interrupt environments.

A process in kernel mode can also put *itself* into an environment similar to the interrupt environment. This action, occurring when the interrupt priority is changed to a priority more favored than INTBASE, can be accomplished by calling the **i_disable** or **disable_lock** kernel service. A kernel-mode process is sometimes required to do this to serialize access to a resource shared by a routine executing in the

interrupt environment. When this is the case, the process operates under most of the same restrictions as a routine executing in the interrupt environment. However, the **e_sleep**, **e_wait**, **e_sleepl**, **et_wait**, **lockl**, and **unlockl** process can sleep, wait, and use locking kernel services if the event word or lock word is pinned.

Routines executed in this environment can adversely affect system real-time performance and are therefore limited to a specific maximum path length. Guidelines for the maximum path length are determined by the interrupt priority at which the routines are executed. Understanding Interrupts provides more information.

# Understanding Kernel Threads

A *thread* is an independent flow of control that operates within the same address space as other independent flows of control within a process.

One process can have multiple threads, with each thread executing different code concurrently, while sharing data and synchronizing much more easily than cooperating processes. Threads require fewer system resources than processes, and can start more quickly.

Although threads can be scheduled, they exist in the context of their process. The following list indicates what is managed at process level and shared among all threads within a process:

- Address space
- System resources, like files or terminals
- Signal list of actions.

The process remains the swappable entity. Only a few resources are managed at thread level, as indicated in the following list:

- State
- Stack
- Signal masks.

**Related concepts**

System Calls
A system call is a routine that allows a user application to request actions that require special privileges. Adding system calls is one of several ways to extend the functions provided by the kernel.

## Kernel Threads, Kernel Only Threads, and User Threads

These are the three kinds of threads.

- Kernel threads
- Kernel-only threads
- User threads.

A *kernel thread* is a kernel entity, like processes and interrupt handlers; it is the entity handled by the system scheduler. A kernel thread runs in user mode environment when executing user functions or library calls; it switches to kernel mode environment when executing system calls.

A *kernel-only thread* is a kernel thread that executes only in kernel mode environment. Kernel-only threads are controlled by the kernel mode environment programmer through kernel services.

User mode programs can access *user threads* through a library (such as the **libpthreads.a** threads library). User threads are part of a portable programming model. User threads are mapped to kernel threads by the threads library, in an implementation dependent manner. The threads library uses a proprietary interface to handle kernel threads. See *Understanding Threads* in *AIX Version 7.1 General Programming Concepts: Writing and Debugging Programs* to get detailed information about the user threads library and their implementation.

All threads discussed in this article are kernel threads; and the information applies only to the kernel mode environment. Kernel threads cannot be accessed from the user mode environment, except through the threads library.

## Kernel Data Structures

The kernel maintains thread- and process-related information in two types of structures.

- The **user** structure contains process-related information.
- The **uthread** structure contains thread-related information.

These structures cannot be accessed directly by kernel extensions and device drivers. They are encapsulated for portability reasons. Many fields that were previously in the **user** structure are now in the **uthread** structure.

## Thread Creation, Execution, and Termination

A process is always created with one thread, called the *initial thread*. The initial thread provides compatibility with previous single-threaded processes. The initial thread's stack is the process stack.

See “Kernel Process Creation, Execution, and Termination” on page 9 to get more information about kernel process creation.

Other threads can be created, using a two-step procedure. The **thread_create** kernel service allocates and initializes a new thread, and sets its state to idle. The **kthread_start** kernel service then starts the thread, using the specified entry point routine.

A thread is terminated when it executes a return from its entry point, or when it calls the **thread_terminate** kernel service. Its resources are automatically freed. If it is the last thread in the process, the process ends.

## Thread Scheduling

Threads are scheduled using one of the scheduling policies that are listed in this section.

- First-in first-out (FIFO) scheduling policy, with fixed priority. Using the FIFO policy with high favored priorities might lead to bad system performance.
- Round-robin (RR) scheduling policy, quantum based and with fixed priority.
- Default scheduling policy, a non-quantum based round-robin scheduling with fluctuating priority. Priority is modified according to the CPU usage of the thread.

Scheduling parameters can be changed using the **thread_setsched** kernel service. The process-oriented **setpri** system call sets the priority of all the threads within a process. The process-oriented **getpri** system call gets the priority of a thread in the process. The scheduling policy and priority of an individual thread can be retrieved from the `ti_policy` and `ti_pri` fields of the **thrdsinfo** structure returned by the **getthrds** system call.

## Thread Signal Handling

The items listed in this section explain the signal handling concepts.

- A signal mask is associated with each thread.
- The list of actions associated with each signal number is shared among all threads in the process.
- If the signal action specifies termination, stop, or continue, the entire process, thus including all its threads, is respectively terminated, stopped, or continued.
- Synchronous signals attributable to a particular thread (such as a hardware fault) are delivered to the thread that caused the signal to be generated.
- Signals can be directed to a particular thread. If the target thread has blocked the signal from delivery, the signal remains pending on the thread until the thread unblocks the signal from delivery, or the action associated with the signal is set to ignore by any thread within the process.

The signal mask of a thread is handled by the **limit_sigs** and **sigsetmask** kernel services. The **kthread_kill** kernel service can be used to direct a signal to a particular thread.

In the kernel environment, when a signal is received, no action is taken (no termination or handler invocation), even for the **SIGKILL** signal. A thread in kernel environment, especially kernel-only threads, must *poll* for signals so that signals can be delivered. Polling ensures the proper kernel-mode serialization. For example, **SIGKILL** will not be delivered to a kernel-only thread that does not poll for signals. Therefore, **SIGKILL** is not necessarily an effective means for terminating a kernel-only thread.

Signals whose actions are applied at generation time (rather than delivery time) have the same effect regardless of whether the target is in kernel or user mode. A kernel-only thread can poll for unmasked signals that are waiting to be delivered by calling the **sig_chk** kernel service. This service returns the signal number of a pending signal that was not blocked or ignored. The thread then uses the signal number to determine which action should be taken. The kernel does not automatically call signal handlers for a thread in kernel mode as it does for user mode.

See "Kernel Process Signal and Exception Handling" on page 10 for more information about signal handling at process level.

# Using Kernel Processes

A kernel process is a process that is created in the kernel protection domain and always executes in the kernel protection domain.

Kernel processes can be used in subsystems, by complex device drivers, and by system calls. They can also be used by interrupt handlers to perform asynchronous processing not available in the interrupt environment. Kernel processes can also be used as device managers where asynchronous input/output (I/O) and device management is required.

**Related concepts**

System Calls

A system call is a routine that allows a user application to request actions that require special privileges. Adding system calls is one of several ways to extend the functions provided by the kernel.

## Introduction to Kernel Processes

A kernel process (kproc) exists only in the kernel protection domain and differs from a user process in the ways listed in this section.

- It is created using the **creatp** and **initp** kernel services.
- It executes only within the kernel protection domain and has all security privileges.
- It can call a restricted set of system calls and all applicable kernel services. For more information, see "System Calls Available to Kernel Extensions" on page 30.
- It has access to the global kernel address space (including the kernel pinned and pageable heaps), kernel code, and static data areas.
- It must poll for signals and can choose to ignore any signal delivered, including a **kill** signal.
- Its text and data areas come from the global kernel heap.
- It cannot use application libraries.
- It has a process-private region containing only the **u-block** (user block) structure and possibly the kernel stack.
- Its *parent process* is the process that issued the **creatp** kernel service to create the process.
- It can change its parent process to the **init** process and can use interrupt disable functions for serialization.
- It can use locking to serialize process-time access to critical data structures.
- It can only be a 64–bit process in the 64–bit kernel.

A kernel process controls directly the kernel threads. Because kernel processes are always in the kernel protection domain, threads within a kernel process are kernel-only threads. For more information on kernel threads, see "Understanding Kernel Threads" on page 6.

A kernel process inherits the environment of its parent process (the one calling the **creatp** kernel service to create it), but with some exceptions. The kernel process does not have a root directory or a current directory when initialized. All uses of the file system functions must specify absolute path names.

Kernel processes created during phase 1 of system boot must not keep any long-term opens on files until phase 2 of system boot or run time has been reached. This is because Base Operating System changes root file systems between phase 1 and phase 2 of system boot. As a result, the system crashes if any files are open at root file system transition time.

## Accessing Data from a Kernel Process

Because kernel processes execute in the more privileged kernel protection domain, a kernel process can access data that user processes cannot.

This applies to all kernel data, of which there are three general categories:

- The **user block** data structure

  The **u-block** (or **u-area)** structure exists for kernel processes and contains roughly the same information for kernel processes as for user-mode processes. A kernel process must use kernel services to query or manipulate data from the **u-area** to maintain modularity and increase portability of code to other platforms.
- The stack for a kernel process

  To ensure binary compatibility with older applications, each kernel process has a stack called the *process stack*. This stack is used by the process initial thread.

  The location of the stack for a kernel process is implementation-dependent. This stack can be located in global memory or in the process-private segment of the kernel process. A kernel process must not assume automatically that its stack is located in global memory.
- Global kernel memory

  A kernel process can also access global kernel memory as well as allocate and de-allocate memory from the kernel heaps. Because it runs in the kernel protection domain, a kernel process can access any valid memory location within the global kernel address space. Memory dynamically allocated from the kernel heaps by the kernel process must be freed by the kernel process before exiting. Unlike user-mode processes, memory that is dynamically allocated by a kernel process is not freed automatically upon process exit.

## Cross-Memory Services

Kernel processes must be provided with a valid cross-memory descriptor to access address regions outside the kernel global address space or kernel process address space. For example, if a kernel process is to access data from a user-mode process, the system call using the process must obtain a cross-memory descriptor for the user-mode region to be accessed.

Calling the **xmattach** or **xmattach64** kernel service provides a descriptor that can then be made available to the kernel process.

The kernel process should then call the **xmemin** and **xmemout** kernel services to access the targeted cross-memory data area. When the kernel process has completed its operation on the memory area, the cross-memory descriptor must be detached by using the **xmdetach** kernel service.

## Kernel Process Creation, Execution, and Termination

A kernel process is created by a kernel-mode routine by calling the **creatp** kernel service.

The **creatp** kernel service allocates and initializes a process block for the process and sets the new process state to idle. This new kernel process does not run until it is initialized by the **initp** kernel service,

which must be called in the same process that created the new kernel process (with the **creatp** service). The **creatp** kernel service returns the process identifier for the new kernel process.

The process is created with one kernel-only thread, called the *initial thread*. See Understanding Kernel Threads to get more information about threads.

After the **initp** kernel service has completed the process initialization, the initial thread is placed on the run queue. On the first dispatch of the newly initialized kernel process, it begins execution at the entry point previously supplied to the **initp** kernel service. The initialization parameters were previously specified in the call to the **initp** kernel service.

A kernel process terminates when it executes a return from its main entry routine. A process should never exit without both freeing all dynamically allocated storage and releasing all locks owned by the kernel process.

When kernel processes exit, the parent process (the one calling the **creatp** and **initp** kernel services to create the kernel process) receives the **SIGCHLD** signal, which indicates the end of a child process. However, it is sometimes undesirable for the parent process to receive the **SIGCHLD** signal due to ending a process. In this case, the kproc can call the **setpinit** kernel service to designate the **init** process as its parent. The **init** process cleans up the state of all its child processes that have become zombie processes. A kernel process can also issue the **setsid** subroutine call to change its session. Signals and job control affecting the parent process session do not affect the kernel process.

## Kernel Process Preemption

A kernel process is initially created with the same process priority as its parent. It can therefore be replaced by a more favored kernel or user process. It does not have higher priority just because it is a kernel process.

Kernel processes can use the **setpri** subroutine to modify their execution priority.

The kernel process can use the locking kernel services to serialize access to critical data structures. This use of locks does not guarantee that the process will not be replaced, but it does ensure that another process trying to acquire the lock waits until the kernel process owning the lock has released it.

Using locks, however, does not provide serialization if a kernel routine can access the critical data while executing in the interrupt environment. Serialization with interrupt handlers must be handled by using locking together with interrupt control. The **disable_lock** and **unlock_enable** kernel services should be used to serialize with interrupt handlers.

Kernel processes must ensure that their maximum path lengths adhere to the specifications for interrupt handlers when executing at an interrupt priority more favored than INTBASE. This ensures that system real-time performance is not degraded.

## Kernel Process Signal and Exception Handling

Signals are delivered to exactly one thread within the process which has not blocked the signal from delivery. If all threads within the target process have blocked the signal from delivery, the signal remains pending on the process until a thread unblocks the signal from delivery, or the action associated with the signal is set to ignore by any thread within the process.

See “Thread Signal Handling” on page 7 for more information on signal handling by threads.

Signals whose action is applied at generation time (rather than delivery time) have the same effect regardless of whether the target is a kernel or user process. A kernel process can poll for unmasked signals that are waiting to be delivered by calling the **sig_chk** kernel service. This service returns the signal number of a pending signal that was not blocked or ignored. The kernel process then uses the signal number to determine which action should be taken. The kernel does not automatically call signal handlers for a kernel process as it does for user processes.

A kernel process should also use the exception-catching facilities (**setjmpx**, and **clrjmpx**) available in kernel mode to handle exceptions that can be caused during run time of the kernel process. Exceptions received during the execution of a kernel process are handled the same as exceptions that occur in any kernel-mode routine.

Unhandled exceptions that occur in kernel mode (in any user process while in kernel mode, in an interrupt handler, or in a kernel process) result in a system crash. To avoid crashing the system due to unhandled exceptions, kernel routines should use the **setjmpx**, **clrjmpx**, and **longjmpx** kernel services to handle exceptions that might possibly occur during run time. See "Understanding Exception Handling" on page 14 for more details on handling exceptions.

### Kernel Process Use of System Calls

System calls made by kernel processes do not result in a change of protection domain because the kernel process is already within the kernel protection domain.

Routines in the kernel (including routines executing in a kernel process) are bound by the loader to the system call function and not to the system call handler. When system calls use kernel services to access user-mode data, these kernel services recognize that the system call is running within a kernel process instead of a user process and correctly handle the data accesses.

However, the error information returned from a kernel process system call must be accessed differently than for a user process. A kernel process must use the **getuerror** kernel service to retrieve the system call error information normally provided in the **errno** global variable for user-mode processes. In addition, the kernel process can use the **setuerror** kernel service to set the error information to 0 before calling the system call. The return code from the system call is handled the same for all processes.

Kernel processes can use only a restricted set of the base system calls. "System Calls Available to Kernel Extensions" on page 30 lists system calls available to kernel processes.

## Accessing User-Mode Data While in Kernel Mode

Kernel extensions must use a set of kernel services to access data that is in the user-mode protection domain. These services ensure that the caller has the authority to perform the desired operation at the time of data access and also prevent system crashes in a system call when accessing user-mode data.

These services can be called only when running in the process environment of the process that contains the user-mode data. For more information on user-mode protection, see "User Protection Domain" on page 21. For more information on the process environment, see "Process Environment" on page 5.

### Data Transfer Services

The lists in this section show user-mode data access kernel services (primitives).

| Kernel Service | Purpose |
|---|---|
| **suword** | Stores a word of data in user memory. |
| **fubyte** | Fetches, or retrieves, a byte of data from user memory. |
| **fuword** | Fetches, or retrieves, a word of data from user memory. |
| **copyin** | Copies data between user and kernel memory. |
| **copyout** | Copies data between user and kernel memory. |
| **copyinstr** | Copies a character string (including the terminating null character) from user to kernel space. |

Additional kernel services allow data transfer between user mode and kernel mode when a **uio** structure is used, thereby describing the user-mode data area to be accessed. Following is a list of services that typically are used between the file system and device drivers to perform device I/O:

| Kernel Service | Purpose |
|---|---|
| **uiomove** | Moves a block of data between kernel space and a space defined by a **uio** structure. |
| **ureadc** | Writes a character to a buffer described by a **uio** structure. |

| Kernel Service | Purpose |
| --- | --- |
| **uwritec** | Retrieves a character from a buffer described by a **uio** structure. |

Kernel services ending in “64”, such as **suword64**, **copyin64** and so on, are deprecated in AIX® 6.1 and later. To maintain binary compatibility of applications, macros in the **sys/uio.h** header file redefine these services to their counterparts when compiling in 64-bit mode.

### Using Cross-Memory Kernel Services

Occasionally, access to user-mode data is required when not in the environment of the user-mode process that has addressability to the data. Such cases occur when the data is to be accessed asynchronously.

Examples of asynchronous accessing include:

- Direct memory access to the user data by I/O devices
- Data access by interrupt handlers
- Data access by a kernel process

In these circumstances, the kernel cross-memory services are required to provide the necessary access. The **xmattach** kernel service allows a cross-memory descriptor to be obtained for the data area to be accessed. These services must be called in the process environment of the process containing the data area.

After a cross-memory descriptor has been obtained, the **xmemin** and **xmemout** kernel services can be used to access the data area outside the process environment containing the data. When access to the data area is no longer required, the access must be removed by calling the **xmdetach** kernel service. Kernel extensions should use these services only when absolutely necessary. Because of the machine dependencies of cross-memory operations, using them increases the difficulty of porting the kernel extension to other machine platforms.

## Understanding Locking

The information in this section is provided to assist you in understanding locking.

**Related concepts**

System Calls

A system call is a routine that allows a user application to request actions that require special privileges. Adding system calls is one of several ways to extend the functions provided by the kernel.

### Lockl Locks

The *lockl locks* (previously called *conventional locks*) are provided for compatibility only and should not be used in new code: simple or complex locks should be used instead. These locks are used to protect a critical section of code which accesses a resource such as a data structure or device, serializing access to the resource. Every thread which accesses the resource must acquire the lock first, and release the lock when finished.

A conventional lock has two states: locked or unlocked. In the *locked* state, a thread is currently executing code in the critical section, and accessing the resource associated with the conventional lock. The thread is considered to be the owner of the conventional lock. No other thread can lock the conventional lock (and therefore enter the critical section) until the owner unlocks it; any thread attempting to do so must wait until the lock is free. In the *unlocked* state, there are no threads accessing the resource or owning the conventional lock.

*Lockl locks* are recursive and, unlike simple and complex locks, can be awakened by a signal.

## Simple Locks

A *simple* lock provides exclusive-write access to a resource such as a data structure or device.

Simple locks are not recursive and have only two states: locked or unlocked.

## Complex Locks

A *complex* lock can provide either shared or exclusive access to a resource such as a data structure or device.

Complex locks are not recursive by default (but can be made recursive) and have three states: exclusive-write, shared-read, or unlocked.

If several threads perform read operations on the resource, they must first acquire the corresponding lock in shared-read mode. Because no threads are updating the resource, it is safe for all to read it. Any thread which writes to the resource must first acquire the lock in exclusive-write mode. This guarantees that no other thread will read or write the resource while it is being updated.

## Types of Critical Sections

There are two types of critical sections which must be protected from concurrent execution in order to serialize access to a resource.

| Item | Description |
|---|---|
| **thread-thread** | These critical sections must be protected (by using the locking kernel services) from concurrent execution by multiple processes or threads. |
| **thread-interrupt** | These critical sections must be protected (by using the **disable_lock** and **unlock_enable** kernel services) from concurrent execution by an interrupt handler and a thread or process. |

## Priority Promotion

When a lower priority thread owns a lock which a higher-priority thread is attempting to acquire, the owner has its priority promoted to that of the most favored thread waiting for the lock.

When the owner releases the lock, its priority is restored to its normal value. Priority promotion ensures that the lock owner can run and release its lock, so that higher priority processes or threads do not remain blocked on the lock.

## Locking Strategy in Kernel Mode

A hierarchy of locks exists. This hierarchy is imposed by software convention, but is not enforced by the system. The lockl **kernel_lock** variable, which is the global kernel lock, has the coarsest granularity. Other types of locks have finer granularity.

**Attention:** A kernel extension should not attempt to acquire the kernel lock if it owns any other lock. Doing so can cause unpredictable results or system failure.

The following list shows the ordering of locks based on granularity:

- The **kernel_lock** global kernel lock

  **Note:** Avoid using the **kernel_lock** global kernel lock variable in new code. It is only included for compatibility purposes.
- File system locks (private to file systems)
- Device driver locks (private to device drivers)
- Private fine-granularity locks

Locks should generally be released in the reverse order from which they were acquired; all locks must be released before a kernel process or thread exits. Kernel mode processes do not receive any signals while they hold any lock.

# Understanding Exception Handling

Exception handling involves a basic distinction between *interrupts* and *exceptions*.

- An interrupt is an asynchronous event and is not associated with the instruction that is executing when the interrupt occurs.
- An exception is a synchronous event and is directly caused by the instruction that is executing when the exception occurs.

The computer hardware generally uses the same mechanism to report both interrupts and exceptions. The machine saves and modifies some of the event's state and forces a branch to a particular location. When decoding the reason for the machine interrupt, the interrupt handler determines whether the event is an interrupt or an exception, then processes the event accordingly.

## Exception Processing

When an exception occurs, the current instruction stream cannot continue. If you ignore the exception, the results of executing the instruction may become undefined. Further execution of the program may cause unpredictable results.

The kernel provides a default exception-handling mechanism by which an instruction stream (a process- or interrupt-level program) can specify what action is to be taken when an exception occurs. Exceptions are handled differently depending on whether they occurred while executing in kernel mode or user mode.

### *Default Exception-Handling Mechanism*

If no exception handler is currently defined when an exception occurs, typically one of two things happens.

- If the exception occurs while a process is executing in user mode, the process is sent a signal relevant to the type of exception.
- If the exception occurs while in kernel mode, the system halts.

## Kernel-Mode Exception Handling

Exception handling in kernel mode extends the **setjump** and **longjump** subroutines context-save-and-restore mechanism.

Exception handling in kernel mode extends these subroutines mechanism by providing **setjmpx** and **longjmpx** kernel services to handle exceptions. The traditional system mechanism is extended by allowing these exception handlers (or context-save checkpoints) to be stacked on a per-process or per-interrupt handler basis.

This stacking mechanism allows the execution point and context of a process or interrupt handler to be restored to a point in the process or interrupt handler, *at the point of return from the* **setjmpx** *kernel service*. When execution returns to this point, the return code from **setjmpx** kernel service indicates the type of exception that occurred so that the process or interrupt handler state can be fully restored. Appropriate retry or recovery operations are then invoked by the software performing the operation.

When an exception occurs, the kernel first-level exception handler gets control. The first-level exception handler determines what type of exception has occurred and saves information necessary for handling the specific type of exception. For an I/O exception, the first-level handler also enables again the programmed I/O operations.

The first-level exception handler then modifies the saved context of the interrupted process or interrupt handler. It does so to execute the **longjmpx** kernel service when the first-level exception handler returns to the interrupted process or interrupt handler.

The **longjmpx** kernel service executes in the environment of the code that caused the exception and restores the current context from the topmost jump buffer on the stack of saved contexts. As a result, the state of the process or interrupt handler that caused the exception is restored to the point of the

return from the **setjmpx** kernel service. (The return code, nevertheless, indicates that an exception has occurred.)

The process or interrupt handler software should then check the return code and invoke exception handling code to restore fully the state of the process or interrupt handler. Additional information about the exception can be obtained by using the **getexcept** kernel service.

### *User-Defined Exception Handling*

An exception handler should perform any necessary clean-up such as freeing storage or segment registers and releasing other resources.

Other than performing necessary clean-up, a typical exception handler should do the following:

- If the exception is recognized by the current handler and can be handled entirely within the routine, the handler should establish itself again by calling the **setjmpx** kernel service. This allows normal processing to continue.
- If the exception is not recognized by the current handler, it must be passed to the previously stacked exception handler. The exception is passed by calling the **longjmpx** kernel service, which either calls the previous handler (if any) or takes the system's default exception-handling mechanism.
- If the exception is recognized by the current handler but cannot be handled, it is treated as though it is unrecognized. The **longjmpx** kernel service is called, which either passes the exception along to the previous handler (if any) or takes the system default exception-handling mechanism.

When a kernel routine that has established an exception handler completes normally, it must remove its exception handler from the stack (by using the **clrjmpx** kernel service) before returning to its caller.

**Note:** When the **longjmpx** kernel service invokes an exception handler, that handler's entry is automatically removed from the stack.

## Implementing Kernel Exception Handlers

The following information is provided to assist you in implementing kernel exception handlers.

### *setjmpx, longjmpx, and clrjmpx Kernel Services*

The **setjmpx** kernel service provides a way to save the following portions of the program state at the point of a call:

- Nonvolatile general registers
- Stack pointer
- TOC pointer
- Interrupt priority number (**intpri**)
- Ownership of kernel-mode lock

This state can be restored later by calling the **longjmpx** kernel service, which accomplishes the following tasks:

- Reloads the registers (including TOC and stack pointers)
- Enables or disables to the correct interrupt level
- Conditionally acquires or releases the kernel-mode lock
- Forces a branch back to the point of original return from the **setjmpx** kernel service

The **setjmpx** kernel service takes the address of a jump buffer (a **label_t** structure) as an explicit parameter. This structure can be defined anywhere including on the stack (as an automatic variable). After noting the state data in the jump buffer, the **setjmpx** kernel service pushes the buffer onto the top of a stack that is maintained in the machine-state save structure.

The **longjmpx** kernel service is used to return to the point in the code at which the **setjmpx** kernel service was called. Specifically, the **longjmpx** kernel service returns to the most recently created jump buffer, as indicated by the top of the stack anchored in the machine-state save structure.

The parameter to the **longjmpx** kernel service is an exception code that is passed to the resumed program as the return code from the **setjmp** kernel service. The resumed program tests this code to determine the conditions under which the **setjmpx** kernel service is returning. If the **setjmpx** kernel service has just saved its jump buffer, the return code is 0. If an exception *has* occurred, the program is entered by a call to the **longjmpx** kernel service, which passes along a return code that is *not* equal to 0.

**Note:** Only the resources listed here are saved by the **setjmpx** kernel service and restored by the **longjmpx** kernel service. Other resources, in particular segment registers, are not restored. A call to the **longjmpx** kernel service, by definition, returns to an earlier point in the program. The program code must free any resources that are allocated between the call to the **setjmpx** kernel service and the call to the **longjmpx** kernel service.

If the exception handler stack is empty when the **longjmpx** kernel service is issued, there is no place to jump to and the system default exception-handling mechanism is used. If the stack is not empty, the context that is defined by the topmost jump buffer is reloaded and resumed. The topmost buffer is then removed from the stack.

The **clrjmpx** kernel service removes the top element from the stack as placed there by the **setjmpx** kernel service. The caller to the **clrjmpx** kernel service is expected to know exactly which jump buffer is being removed. This should have been established earlier in the code by a call to the **setjmpx** kernel service. Accordingly, the address of the buffer is required as a parameter to the **clrjmpx** kernel service. It can then perform consistency checking by asserting that the address passed is indeed the address of the top stack element.

### *Exception Handler Environment*

The stacked exception handlers run in the environment in which the exception occurs. That is, an exception occurring in a process environment causes the next dispatch of the process to run the exception handler on the top of the stack of exception handlers for that process.

An exception occurring in an interrupt handler causes the interrupt handler to return to the context saved by the last call to the **setjmpx** kernel service made by the interrupt handler.

**Note:** An interrupt handler context is newly created each time the interrupt handler is invoked. As a result, exception handlers for interrupt handlers must be registered (by calling the **setjmpx** kernel service) each time the interrupt handler is invoked. Otherwise, an exception detected during execution of the interrupt handler will be handled by the default handler.

### *Restrictions on Using the setjmpx Kernel Service*

Process and interrupt handler routines registering exception handlers with the **setjmpx** kernel service must not return to their caller before removing the saved jump buffer or buffers from the list of jump buffers.

A saved jump buffer can be removed by invoking the **clrjmpx** kernel service in the reverse order of the **setjmpx** calls. The saved jump buffer must be removed before return because the routine's context no longer exists once the routine has returned to its caller.

If, on the other hand, an exception does occur (that is, the return code from **setjmpx** kernel service is nonzero), the jump buffer is automatically removed from the list of jump buffers. In this case, a call to the **clrjmpx** kernel service for the jump buffer must not be performed.

Care must also be taken in defining variables that are used after the context save (the call to the **setjmpx** service), and then again by the exception handler. Sensitive variables of this nature must be restored to their correct value by the exception handler when an exception occurs.

**Note:** If the last value of the variable is desired at exception time, the variable data type must be declared as "volatile."

Exception handling is concluded in one of two ways. Either a registered exception handler handles the exception and continues from the saved context, or the default exception handler is reached by exhausting the stack of jump buffers.

### *Exception Codes*

The **/usr/include/sys/except.h** file contains a list of code numbers corresponding to the various types of hardware exceptions. When an exception handler is invoked (the return from the **setjmpx** kernel service is not equal to 0), it is the responsibility of the handler to test the code to ensure that the exception is one the routine can handle.

If it is not an expected code, the exception handler must:

- Release any resources that would not otherwise be freed (buffers, segment registers, storage acquired using the **xmalloc** routines)
- Call the **longjmpx** kernel service, passing it the exception code as a parameter

Thus, when an exception handler does not recognize the exception for which it has been invoked, it passes the exception on to the next most recent exception handler. This continues until an exception handler is reached that recognizes the code and can handle it. Eventually, if no exception handler can handle the exception, the stack is exhausted and the system default action is taken.

In this manner, a component can allocate resources (after calling the **setjmpx** kernel service to establish an exception handler) and be assured that the resources will later be released. This ensures the exception handler gets a chance to release those resources regardless of what events occur before the instruction stream (a process- or interrupt-level code) is terminated.

By coding the exception handler to recognize what exception codes it can process rather than encoding this knowledge in the stack entries, a powerful and simple-to-use mechanism is created. Each handler need only investigate the exception code that it receives rather than just assuming that it was invoked because a particular exception has occurred to implement this scheme. The set of exception codes used cannot have duplicates.

Exceptions generated by hardware use one of the codes in the **/usr/include/sys/except.h** file. However, the **longjmpx** kernel service can be invoked by any kernel component, and any integer can serve as the exception code. A mechanism similar to the old-style **setjmp** and **longjmp** kernel services can be implemented on top of the **setjmpx**/**longjmpx** stack by using exception codes outside the range of those used for hardware exceptions.

To implement this old-style mechanism, a unique set of exception codes is needed. These codes must not conflict with either the pre-assigned hardware codes or codes used by any other component. A simple way to get such codes is to use the addresses of unique objects as code values.

For example, a program that establishes an exception handler might compare the exception code to the address of its own entry point. Later on in the calling sequence, after any number of intervening calls to the **setjmpx** kernel service by other programs, a program can issue a call to the **longjmpx** kernel service and pass the address of the agreed-on function descriptor as the code. This code is only recognized by a single exception handler. All the intervening ones just clean up their resources and pass the code to the **longjmpx** kernel service again.

Addresses of functions are not the only possibilities for unique code numbers. For example, addresses of external variables can also be used. By using unigue, system-wide addresses, the problem of code-space collision is transformed into a problem of external-name collision. This problem is easier to solve, and is routinely solved whenever the system is built. By comparison, pre-assigning exception numbers by using **#define** statements in a header file is a much more cumbersome and error-prone method.

### *Hardware Detection of Exceptions*

Each of the exception types results in a hardware interrupt. For each such interrupt, a first-level interrupt handler (FLIH) saves the state of the executing program and calls a second-level handler (SLIH). The SLIH is passed a pointer to the machine-state save structure and a code indicating the cause of the interrupt.

When a SLIH determines that a hardware interrupt should actually be considered a synchronous exception, it sets up the machine-state save to invoke the **longjmpx** kernel service, and then returns. The FLIH then resumes the instruction stream at the entry to the **longjmpx** service.

The **longjmpx** service then invokes the top exception handler on the stack or takes the system default action as previously described.

### User-Mode Exception Handling

Exceptions that occur in a user-mode process and are not automatically handled by the kernel cause the user-mode process to be signaled. If the process is in a state in which it cannot take the signal, it is terminated and the information logged. Kernel routines can install user-mode exception handlers that catch exceptions before they are signaled to the user-mode process.

The **uexadd** and **uexdel** kernel services allow system-wide user-mode exception handlers to be added and removed.

The most recently registered exception handler is the first called. If it cannot handle the exception, the next most recent handler on the list is called, and this second handler attempts to handle the exception. If this attempt fails, successive handlers are tried, until the default handler is called, which generates the signal.

Additional information about the exception can be obtained by using the **getexcept** kernel service.

## Using Kernel Extensions for 64–bit Processes

Kernel extensions in a 64-bit kernel run in 64-bit mode. Therefore, only 64-bit kernel extensions can run on AIX 6.1 and later. You can program kernel extensions for both 32-bit and 64-bit applications.

System calls can be made available to 32- or 64-bit processes, selectively. If an application invokes a system call that is not exported to processes running in the current mode, the call fails.

Because only 64-bit kernel extensions can run on AIX® 6.1 and later, the interaction between kernel extensions and user address space is simplified. To examine and manipulate user address space, use kernel services such as the **as_att64**, **as_det64**, **as_puth64**, **as_seth64**, and **as_getsrval64** kernel services.

Address space remapping is no longer necessary for kernel extensions running on AIX® 6.1 and later. 64-bit pointers or 64-bit data values of the **long** type can be used directly by the kernel without manipulation because the kernel always runs in 64-bit mode. The 64-bit kernel can also handle 64-bit addresses without mapping them to 32-bit values.

**32-bit:** Do not use the **as_att**, **as_det**, **as_seth**, **as_geth**, **as_puth**, and **as_getsrval** kernel services; they are obsolete. The **as_remap64**, **as_unremap64**, **get64bitparm**, and **saveretval64** kernel services are also obsolete.

## 64-bit Kernel Extension Programming Environment

The concepts included in this section explain the 64-bit kernel extension programming environment.

### C Language Data Model

The 64-bit kernel uses the LP64 (Long Pointer 64-bit) C language data model and requires kernel extensions to do the same. The LP64 data model defines **pointers**, **long**, and **long long** types as 64 bits, **int** as 32 bits, **short** as 16 bits, and **char** as 8 bits.

In the ILP32 data model, **long** and **pointer** types are 32 bits. In order to port an existing 32-bit kernel extension to the 64-bit kernel environment, source code must be modified to be type-safe under LP64. This means ensuring that data types are used in a consistent fashion. Source code is incorrect for the 64-bit environment if it assumes that pointers, **long**, and **int** are all the same size.

In addition, the use of system-derived types must be examined whenever values are passed from an application to the kernel. For example, **size_t** is a system-derived type whose size depends on the compilation mode, and **key_t** is a system-derived type that is 64 bits in the 64-bit kernel environment.

## Kernel Data Structures

Several global, exported kernel data structures have been changed in the 64-bit kernel, in order to support scalability and future functionality.

These changes include larger structure sizes as a result of being compiled under the LP64 data model. In porting a kernel extension to the 64-bit kernel environment, these data structure changes must be considered.

## Function Prototypes

Function prototypes are more important in the 64-bit programming environment than the 32-bit programming environment, because the default return value of an undeclared function is **int**.

If a function prototype is missing for a function returning a pointer, the compiler will convert the returned value to an **int** by setting the high-order word to 0, corrupting the value. In addition, function prototypes allow the compiler to do more type checking, regardless of the compilation mode.

When compiled in 64-bit mode, system header files define full function prototypes for all kernel services provided by the 64-bit kernel. By defining the **__FULL_PROTO** macro, function prototypes are provided in 32-bit mode as well. It is recommended that function prototypes be provided by including the system header files, instead of providing a prototype in a source file.

## Compiler Options

To compile a kernel extension in 64-bit mode, the **-q64** flag must be used.

To check for missing function prototypes, **-qinfo=pro** can be specified. To compile in ANSI mode, use the **-qlanglvl=ansi** flag. When this flag is used, additional error checking will be performed by the compiler. To link-edit a kernel extension, the **-b64** option must be used with the **ld** command.

**Note:** Do not link kernel extensions using the **cc** command.

## Conditional Compilation

When compiling in 64-bit mode, the compiler automatically defines the macro **__64BIT__**. Kernel extensions should always be compiled with the **_KERNEL** macro defined, and if **sys/types.h** is included, the macro **__64BIT_KERNEL** will be defined for kernel extensions being compiled in 64-bit mode. The **__64BIT_KERNEL** macro can be used to provide for conditional compilation when compiling kernel extensions from common source code.

Kernel extensions should not be compiled with the **_KERNSYS** macro defined. If this macro is defined, the resulting kernel extension will not be supported, and binary compatibility will not be assured with future releases.

## Kernel Extension Libraries

The **libcsys.a** and **libsys.a** libraries are supported for kernel extensions.

Function prototypes for all the functions in **libcsys.a** are found in **sys/libcsys.h**.

## Kernel Execution Mode

Within the 64-bit kernel, all kernel mode subsystems, including kernel extensions, run exclusively in 64-bit processor mode and are capable of accessing data or executing instructions at any location within the kernel's 64-bit address space, including those found above the first 4GBs of this address space.

This availability of the full 64-bit address space extends to all kernel entities, including kprocs and interrupt handlers, and enables the potential for software resource scalability through the introduction of an enormous kernel address space.

### Kernel Address Space

The 64-bit kernel provides a common user and kernel 64-bit address space.

#### *Kernel Address Space Organization*

The organization of kernel space differs between hardware systems.

Therefore, kernel extensions must not have any dependencies on the locations, relative or absolute, of the kernel text, kernel global data, kernel heap data, and kernel stack values, and must appropriately type variables used to hold kernel addresses.

#### *Temporary Attachment*

The 64-bit kernel provides kernel extensions with the capability to temporarily attach virtual memory segments to the kernel space for the current thread of kernel mode execution. This capability is provided through the **vm_att** and **vm_det** services.

A total of four concurrent temporary attaches will be supported under a single thread of execution.

#### *Global Regions*

The 64-bit kernel provides kernel extensions with the capability to create global regions within the kernel address space. Once created, a region is globally accessible to all kernel code until it is destroyed. Regions may be created with unique characteristics, for example, page protection, that suit kernel extension requirements and are different from the global virtual memory allocated from the kernel_heap.

Global regions are also useful for kernel extensions that in the past have organized their data around virtual memory segments and require sizes and alignments that are inappropriate for the kernel heap. Under the 64-bit kernel, this memory can be provided through global regions rather than separate virtual memory segments, thus avoiding the complexity and performance cost of temporarily attaching virtual memory segments.

Global regions are created and destroyed with the **vm_galloc** and **vm_gfree** kernel services.

## System Calls

A system call is a routine that allows a user application to request actions that require special privileges. Adding system calls is one of several ways to extend the functions provided by the kernel.

The distinction between a system call and an ordinary function call is only important in the kernel programming environment. User-mode application programs are not usually aware of this distinction.

Operating system functions are made available to the application program in the form of *programming libraries*. A set of library functions found in a library such as **libc.a** can have functions that perform some user-mode processing and then internally start a system call. In other cases, the system call can be directly exported by the library without any user-space code. For more information on programming libraries, see “Using Libraries” on page 4.

Operating system functions available to application programs can be split or moved between user-mode functions and kernel-mode functions as required for different releases or machine platforms. Such movement does not affect the application program. “Kernel Environment” on page 1 provides more information on how to use system calls in the kernel environment.

Review the descriptions of the following subroutine and kernel services:

- **fork** subroutine
- **e_sleep** kernel service
- **e_sleepl** kernel service
- **et_wait** kernel service
- **getuerror** kernel service
- **initp** kernel service
- **lockl** kernel service

- **longjmpx** kernel service
- **setjmpx** kernel service
- **setuerror** kernel service
- **unlockl** kernel service

**Related concepts**

Understanding Kernel Threads
A *thread* is an independent flow of control that operates within the same address space as other independent flows of control within a process.

Using Kernel Processes
A kernel process is a process that is created in the kernel protection domain and always executes in the kernel protection domain.

Understanding Locking
The information in this section is provided to assist you in understanding locking.

Locking Kernel Services
The kernel services can be locked using any of the methods that are described in this overview.

Understanding Interrupts
Each hardware interrupt has an interrupt level, trigger, and interrupt priority.

**Related reference**

Using Libraries
The operating system provides libraries that can be used by kernel extensions.

## Differences Between a System Call and a User Function

A system call differs from a user function in several key ways.

- A system call has more privilege than a normal subroutine. A system call runs with kernel-mode privilege in the kernel protection domain.
- System call code and data are located in global kernel memory.
- System call routines can create and use kernel processes to perform asynchronous processing.
- System calls cannot use shared libraries or any symbols not found in the kernel protection domain.

## Understanding Protection Domains

There are two protection domains in the operating system: the *user protection domain* and the *kernel mode protection domain*.

**Related concepts**

Virtual File Systems
The virtual file system (VFS) interface, also known as the v-node interface, provides a bridge between the physical and logical file systems. The information that follows discusses the virtual file system interface, its data structures, and its header files, and explains how to configure a virtual file system.

### User Protection Domain

Application programs run in the *user protection domain*.

Application programs provide the following types of access:

- Read and write access to the data region of the process
- Read access to the text and shared text regions of the process
- Access to shared data regions using the shared memory functions.

When a program is running in the user protection domain, the processor executes instructions in the problem state, and the program does not have direct access to kernel data.

### Kernel Protection Domain

The code in the kernel and kernel extensions run in the *kernel protection domain*. This code includes interrupt handlers, kernel processes, device drivers, system calls, and file system code.

The processor is in the kernel protection domain when it executes instructions in the privileged state, which provides:

- Read and write access to the global kernel address space
- Read and write access to the thread's **uthread** block and u-block, except when an interrupt handler is running.

Code running in the kernel protection domain can affect the execution environments of all processes because it:

- Can access global system data
- Can use all kernel services
- Is exempt from all security constraints.

Programming errors in the code running in the kernel protection domain can cause the operating system to fail. In particular, a process's user data cannot be accessed directly, but must be accessed using the **copyin** and **copyout** kernel services, or their variants. These routines protect the kernel from improperly supplied user data addresses.

Application programs can gain controlled access to kernel data by making system calls. Access to functions that directly or indirectly invoke system calls is typically provided by programming libraries, providing access to operating system functions.

## Understanding System Call Execution

When a user program invokes a system call, a system call instruction is executed, which causes the processor to begin executing the system call handler in the kernel protection domain.

This system call handler performs the following actions:

1. Sets the `ut_error` field in the **uthread** structure to 0
2. Switches to a kernel stack associated with the calling thread
3. Calls the function that implements the requested system call.

The system loader maintains a table of the functions that are used for each system call.

The system call runs within the calling thread, but with more privilege because system calls run in the kernel protection domain. After the function implementing the system call has performed the requested action, control returns to the system call handler. If the `ut_error` field in the **uthread** structure has a non-zero value, the value is copied to the application's thread-specific **errno** variable. If a signal is pending, signal processing take place, which can result in an application's signal handler being invoked. If no signals are pending, the system call handler restores the state of the calling thread, which is resumed in the user protection domain. For more information on protection domains, see “Understanding Protection Domains” on page 21.

## Accessing Kernel Data While in a System Call

A system call can access data that the calling thread cannot access because system calls execute in the kernel protection domain.

The following are the general categories of kernel data:

- The **ublock** or **u-block** (user block data) structure:

  System calls should use the kernel services to read or modify data traditionally found in the **ublock** or **uthread** structures. For example, the system call handler uses the value of the thread’s `ut_error` field to update the thread-specific **errno** variable before returning to user mode. This field can be read or set by using the **getuerror** and **setuerror** kernel services. The current process ID can be obtained by using

the **getpid** kernel service, and the current thread ID can be obtained by using the **thread_self** kernel service.

- Global memory:

  System calls can also access global memory such as the kernel and kernel data regions. These regions contain the code and static data for the system call as well as the rest of the kernel.

- The stack for a system call:

  A system call routine runs on a protected stack associated with a calling thread, which allows a system call to execute properly even when the stack pointer to the calling thread is invalid. In addition, privileged data can be saved on the stack without danger of exposing the data to the calling thread.

**Attention:** Incorrectly modifying fields in kernel or user block structures can cause unpredictable results or system crashes.

## Passing Parameters to System Calls

Parameters are passed to system calls in the same way that parameters are passed to other functions, but some additional calling conventions and limitations apply.

First, system calls cannot have floating-point parameters. In fact, the operating system does not preserve the contents of floating-point registers when a system call is preempted by another thread, so system calls cannot use any floating-point operations.

Second, because a system call runs on its own stack, the number of arguments that can be passed to a system call is limited. The operating system linkage conventions specify that up to eight general purpose registers are used for parameter passing. If more parameters exist than will fit in eight registers, the remaining parameters are passed in the stack. Because a system call does not have direct access to the application's stack, all parameters for system calls must fit in eight registers.

Third, some parameters are passed in multiple registers. For example, 32-bit applications pass **long long** parameters in two registers, and structures passed by value can require multiple registers, depending on the structure size. The writer of a system call should be familiar with the way parameters are passed by the compiler and ensure that the 8-register limit is not exceeded. For more information on parameter calling conventions, see Subroutine Linkage Convention.

Finally, because 32-bit applications are supported by the 64-bit kernel, the data model used by the kernel does not always match the data model used by the application. When the data models do not match, the system call might have to perform extra processing before parameters can be used.

The **IS64U** macro can be used to determine if the caller of a system call is a 64-bit process.

The ILP32 and LP64 data models differ in the way that pointers and **long** and **long long** parameters are treated when used in structures or passed as functional parameters. The following tables summarize the differences.

| Type | Size | Used as Parameter |
|---|---|---|
| **long** | 32 bits | One register |
| **pointer** | 32 bits | One register |
| **long long** | 64 bits | Two registers |

| Type | Size | Used as Parameter |
|---|---|---|
| **long** | 64 bits | One register |
| **pointer** | 64 bits | One register |
| **long long** | 64 bits | One register |

System calls using these types must take the differing data models into account. The treatment of these types depends on whether they are used as parameters or in structures passed as parameters by value or by reference.

## Passing Scalar Parameters to System Calls

The treatment of these types depends on whether they are used as parameters or in structures passed as parameters by value or by reference.

### *32-bit Application Support on the 64-bit Kernel*

When a 32-bit application makes a system call to the 64-bit kernel, the system call handler zeros the high-order word of each parameter register.

This allows 64-bit system calls to use pointers and unsigned **long** parameters directly. Signed and unsigned integer parameters can also be used directly by 64-bit system calls. This is because in 64-bit mode, the compiler generates code that sign extends or zero fills integers passed as parameters. Similar processing is performed for **char** and **short** parameters, so these types do not require any special handling either. Only signed **long** and **long long** parameters need additional processing.

*Signed long Parameter*

To convert a 32-bit signed **long** parameter to a 64-bit value, the 32-bit value must be sign extended.

The **LONG32TOLONG64** macro is provided for this operation. It converts a 32-bit signed value into a 64-bit signed value, as shown in this example:

```
syscall1(long incr)
    {
        /* If the caller is a 32-bit process, convert
         * 'incr' to a signed, 64-bit value.
         */
        if (!IS64U)
            incr = LONG32TOLONG64(incr);
        .
        .
        .
    }
```

If a parameter can be either a pointer or a symbolic constant, special handling is needed. For example, if `-1` is passed as a pointer argument to indicate a special case, comparing the pointer to `-1` will fail, as will unconditionally sign-extending the parameter value. Code similar to the following should be used:

```
syscall2(void *ptr)
    {
        /* If caller is a 32-bit process,
         * check for special parameter value.
         */
        if (!IS64U && (LONG32TOLONG64(ptr) == -1)
                  ptr = (void *)-1;

        if (ptr == (void *)-1)
            special_handling();
        else {
            .
            .
            .
        }
    }
```

Similar treatment is required when an unsigned long parameter is interpreted as a signed value.

*long long Parameters*

A 32-bit application passes a **long long** parameter in two registers, while a 64-bit kernel system call uses a single register for a **long long** parameter value.

The system call function prototype cannot match the function prototype used by the application. Instead, each **long long** parameter should be replaced by a pair of **uintptr_t** parameters. Subsequent parameters

should be replaced with **uintptr_t** parameters as well. When the caller is a 32-bit process, a single 64-bit value will be constructed from two consecutive parameters. This operation can be performed using the **INTSTOLLONG** macro. For a 64-bit caller, a single parameter is used directly.

For example, suppose the application function prototype is:

```
syscall3(void *ptr, long long len1, long long len2, int size);
```

The corresponding system call code should be similar to:

```
syscall3(void *ptr, uintptr_t L1,
             uintptr_t L2, uintptr_t L3,
             uintptr_t L4, uintptr_t L5)
    {
        long len1;
       long len2;
       int size;

        /* If caller is a 32-bit application, len1
         * and len2 must be constructed from pairs of
         * parameters.  Otherwise, a single parameter
         * can be used for each length.
         */

        if (!IS64U) {
            len1 = INTSTOLLONG(L1, L2);
            len2 = INTSTOLLONG(L3, L4);
            size = (int)L5;
        }
        else {
            len1 = (long)L1
            len2 = (long)L2
            size = (int)L3;
        }
        .
        .
        .
    }
```

## 64-bit Application Support on the 64-bit Kernel

For the most part, system call parameters from a 64-bit application can be used directly by 64-bit system calls.

The system call handler does not modify the parameter registers, so the system call sees the same values that were passed by the application. The only exceptions are the **pid_t** and **key_t** types, which are 32-bit signed types in 64-bit applications, but are 64-bit signed types in 64-bit system calls. Before these two types can be used, the 32-bit parameter values must be sign extended using the **LONG32TOLONG64** macro.

## Passing Structure Parameters to System Calls

When structures are passed to or from system calls, whether by value or by reference, the layout of the structure in the application might not match the layout of the same structure in the system call.

There are two ways that system calls can process structures passed from or to applications: structure reshaping and dual implementation.

### *Structure Reshaping*

Structure reshaping allows system calls to support both 32- and 64-bit applications using a single system call interface and using code that is predominately common to both application types.

Structure reshaping requires defining more than one version of a structure. One version of the structure is used internally by the system call to process the request. The other version should use size-invariant types, so that the layout of the structure fields matches the application's view of the structures. When a structure is copied in from user space, the application-view structure definition is used. The structure

is reshaped by copying each field of the application's structure to the kernel's structure, converting the fields as required. A similar conversion is performed on structures that are being returned to the caller.

Structure reshaping is used for structures whose size and layout as seen by an application differ from the size and layout as seen by the system call. If the system call uses a structure definition with fields big enough for both 32- and 64-bit applications, the system call can use this structure, independent of the mode of the caller.

While reshaping requires two versions of a structure, only one version is public and visible to the end user. This version is the natural structure, which can also be used by the system call if reshaping is not needed. The private version should only be defined in the source file that performs the reshaping. The following example demonstrates the techniques for passing structures to system calls that are running in the 64-bit kernel and how a structure can be reshaped:

```
/* Public definition */
struct foo {
    int a;
    long b;
};

/* Private definition--matches 32-bit
 * application's view of the data structure.  */
struct foo32 {
    int a;
    int b;
}

syscall7(struct foo *f)
{
    struct foo   f1;
    struct foo32 f2;

    if (IS64U()) {
        copyin(&f1, f, sizeof(f1));
    }
    else {
        copyin(&f2, f, sizeof(f2));
        f1.a = f2.a;
        f1.b = f2.b;
    }
    /* Common structure f1 used from now on. */
    .
    .
    .
}
```

*Dual Implementation*
The dual implementation approach involves separate code paths for calls from 32-bit applications and calls from 64-bit applications.

Similar to reshaping, the system call code defines a private view of the application's structure. With dual implementations, the function *syscall7* could be rewritten as:

```
syscall8(struct foo *f)
{
     struct foo   f1;
    struct foo32 f2;

    if (IS64U()) {
        copyin(&f1, f, sizeof(f1));
        /* Code for 64-bit process uses f1 */
        .
        .
        .
    }
    else {
        copyin(&f2, f, sizeof(f2));
        /* Code for 32-bit process uses f2 */
        .
        .
        .
```

```
    }
}
```

Dual implementation is most appropriate when the structures are so large that the overhead of reshaping would affect the performance of the system call.

*Passing Structures by Value*
When structures are passed by value, the structure is loaded into as many parameter registers as are needed. When the data model of an application and the data model of the kernel extension differ, the values in the registers cannot be used directly. Instead, the registers must be stored in a temporary variable.

For example:

**Note:** This example builds upon the structure definitions defined in "Dual Implementation" on page 26.

```
/* Application prototype:  syscall9(struct foo f); */

syscall9(unsigned long a1, unsigned long a1)
{
       union {
              struct foo   f1;      /* Structure for 64-bit caller. */
              struct foo32 f2;      /* Structure for 32-bit caller. */
              unsigned long p64[2]; /* Overlay for parameter registers
                                       * when caller is 64-bit program
                                    */
              unsigned int  p32[2]; /* Overlay for parameter registers
                                       * when caller is 32-bit program
                                    */
       } uarg;
       if (IS64U()) {
              uarg.p64[0] = a1;
              uarg.p64[1] = a2;
              /* Now uarg.f1 can be used */
              .
              .
              .
       }
       else {
              uarg.p32[0] = a1;
              uarg.p32[1] = a2;
              /* Now uarg.f2 can be used */
              .
              .
              .
       }
}
```

## Preempting a System Call

The kernel allows a thread to be preempted by a more favored thread, even when a system call is executing. This capability provides better system responsiveness for large multi-user systems.

Because system calls can be preempted, access to global data must be serialized. Kernel locking services, such as **simple_lock** and **simple_unlock**, are frequently used to serialize access to kernel data. A thread can be preempted even when it owns a lock. If multiple locks are obtained by system calls, a technique must be used to prevent multiple threads from deadlocking. One technique is to define a lock hierarchy. A system call must never return while holding a lock. For more information on locking, see "Understanding Locking" on page 12.

## Handling Signals While in a System Call

Signals can be generated asynchronously or synchronously with respect to the thread that receives the signal. An asynchronously generated signal is one that results from some action external to a thread. It

is not directly related to the current instruction stream of that thread. Generally these are generated by other threads or by device drivers.

A synchronously generated signal is one that results from the current instruction stream of the thread. These signals cause interrupts. Examples of such cases are the execution of an illegal instruction, or an attempted data access to nonexistent address space.

**Related concepts**

Kernel Environment
The kernel is dynamically extendable and can be expanded by adding routines that belong to any of the functional classes.

Kernel Services
*Kernel services* are routines that provide the runtime kernel environment to programs executing in kernel mode. Kernel extensions call kernel services, which resemble library routines. In contrast, application programs call library routines.

## Delivery of Signals to a System Call

Delivery of signals to a thread only takes place when a user application is about to be resumed in the user protection domain.

Signals cannot be delivered to a thread if the thread is in the middle of a system call. For more information on signal delivery for kernel processes, see “Using Kernel Processes” on page 8.

## Asynchronous Signals and Wait Termination

An asynchronous signal can alter the operation of a system call or kernel extension by terminating a long wait.

Kernel services such as **e_block_thread**, **e_sleep_thread**, and **et_wait** are affected by signals. The following options are provided when a signal is posted to a thread:

- Return from the kernel service with a return code indicating that the call was interrupted by a signal
- Call the **longjmpx** kernel service to resume execution at a previously saved context in the event of a signal
- Ignore the signal using the **short-wait** option, allowing the kernel service to return normally.

The **sleep** kernel service, provided for compatibility, also supports the **PCATCH** and **SWAKEONSIG** options to control the response to a signal during the **sleep** function.

Previously, the kernel automatically saved context on entry to the system call handler. As a result, any long (interruptible) sleep not specifying the **PCATCH** option returned control to the saved context when a signal interrupted the wait. The system call handler then set the **errno** global variable to **EINTR** and returned a return code of -1 from the system call.

The kernel, however, requires each system call that can directly or indirectly issue a **sleep** call without the **PCATCH** option to set up a saved context using the **setjmpx** kernel service. This is done to avoid overhead for system calls that handle waits terminated by signals. Using the **setjmpx** service, the system can set up a saved context, which sets the system call return code to a -1 and the `ut_error` field to **EINTR**, if a signal interrupts a long wait not specifying **return-from-signal**.

It is probably faster and more robust to specify **return-from-signal** on all long waits and use the return code to control the system call return.

## Stacking Saved Contexts for Nested setjmpx Calls

The kernel supports nested calls to the **setjmpx** kernel service.

The **setjmpx** kernel service implements the stack of saved contexts by maintaining a linked list of context information anchored in the machine state save area. This area is in the user block structure for a process. Interrupt handlers have special machine state save areas.

An initial context is set up for each process by the **initp** kernel service for kernel processes and by the **fork** subroutine for user processes. The process terminates if that context is resumed.

## Handling Exceptions While in a System Call

Exceptions are detected by the processor as a result of the current instruction stream.

Exceptions are interrupts, therefore they take effect synchronously with respect to the current thread.

The default exception handler generates a signal if the process is in a state where signals can be delivered immediately. Otherwise, the default exception handler generates a system dump.

### Alternative Exception Handling Using the setjmpx Kernel Service

Each exception handler is passed the exception type as a parameter.

For certain types of exceptions, a system call can specify unique exception-handler routines through calls to the **setjmpx** service. The exception handler routine is saved as part of the stacked saved context.

The exception handler returns a value that can specify any of the following:

- The process should resume with the instruction that caused the exception.
- The process should return to the saved context that is on the top of the stack of contexts.
- The exception handler did not handle the exception.

If the exception handler did not handle the exception, then the next exception handler in the stack of contexts is called. If none of the stacked exception handlers handle the exception, the kernel performs default exception handling. The **setjmpx** and **longjmpx** kernel services help implement exception handlers.

## Understanding Nesting and Kernel-Mode Use of System Calls

The operating system supports nested system calls with some restrictions. System calls (and any other kernel-mode routines running under the process environment of a user-mode process) can use system calls that pass all parameters by value.

System calls and other kernel-mode routines must not start system calls that have one or more parameters passed by reference. Doing so can result in a system crash. This is because system calls with reference parameters assume that the referenced data area is in the user protection domain. As a result, these system calls must use special kernel services to access the data. However, these services are unsuccessful if the data area they are trying to access is not in the user protection domain.

This restriction does not apply to kernel processes. User-mode data access services can distinguish between kernel processes and user-mode processes in kernel mode. As a result, these services can access the referenced data areas accessed correctly when the caller is a kernel process.

Kernel processes cannot call the **fork** or **exec** system calls, among others. A list of the base operating system calls available to system calls or other routines in kernel mode is provided in “System Calls Available to Kernel Extensions” on page 30.

## Page Faulting within System Calls

Most data accessed by system calls is pageable by default. This includes the system call code, static data, dynamically allocated data, and stack. As a result, a system call can be preempted in two ways:

**Attention:** A page fault that occurs while external interrupts are disabled results in a system crash. Therefore, a system call should be programmed to ensure that its code, data, and stack are pinned before it disables external interrupts.

- By a more favored process, or by an equally favored process when a time slice has been exhausted
- By losing control of the processor when it page faults

In the latter case, even less-favored processes can run while the system call is waiting for the paging I/O to complete.

## Returning Error Information from System Calls

Error information returned by system calls differs from that returned by kernel services that are not system calls. System calls typically return a special value, such as `-1` or `NULL`, to indicate that an error has occurred.

When an error condition is to be returned, the `ut_error` field should be updated by the system call before returning from the system call function. The `ut_error` field is written using the **setuerror** kernel service.

Before actually calling the system call function, the system call handler sets the `ut_error` field to 0. Upon return from the system call function, the system call handler copies the value found in `ut_error` into the thread-specific **errno** variable if `ut_error` was nonzero. After setting the **errno** variable, the system call handler returns to user mode with the return code provided by the system call function.

Kernel-mode callers of system calls must be aware of this return code convention and use the **getuerror** kernel service to obtain the error value when an error indication is returned by the system call. When system calls are nested, the system call function called by the system call handler can return the error value provided by the nested system call function or can replace this value with a new one by using the **setuerror** kernel service.

## System Calls Available to Kernel Extensions

The system calls are grouped according to which subroutines call them.

**Note:** System calls are not available to interrupt handlers.

**Related concepts**

Kernel Environment
The kernel is dynamically extendable and can be expanded by adding routines that belong to any of the functional classes.

Kernel Services
*Kernel services* are routines that provide the runtime kernel environment to programs executing in kernel mode. Kernel extensions call kernel services, which resemble library routines. In contrast, application programs call library routines.

### System Calls Available to All Kernel Extensions

The following section lists the system calls that are available to all kernel extensions.

| Item | Description |
|---|---|
| **gethostid** | Gets the unique identifier of the current host. |
| **getpgrp** | Gets the process ID, process group ID, and parent process ID. |
| **getppid** | Gets the process ID, process group ID, and parent process ID. |
| **getpri** | Returns the scheduling priority of a process. |
| **getpriority** | Gets or sets the *nice* value. |
| **semget** | Gets a set of semaphores. |
| **seteuid** | Sets the process user IDs. |
| **setgid** | Sets the process group IDs. |
| **sethostid** | Sets the unique identifier of the current host. |
| **setpgid** | Sets the process group IDs. |
| **setpgrp** | Sets the process group IDs. |

| Item | Description |
|---|---|
| **setpri** | Sets a process scheduling priority to a constant value. |
| **setpriority** | Gets or sets the *nice* value. |
| **setreuid** | Sets the process user IDs. |
| **setsid** | Creates a session and sets the process group ID. |
| **setuid** | Sets the process user IDs. |
| **ulimit** | Sets and gets user limits. |
| **umask** | Sets and gets the value of the file-creation mask. |

## System Calls Available to Kernel Processes Only

The following section lists the system calls that are available to kernel processes only.

| Item | Description |
|---|---|
| **disclaim** | Disclaims the content of a memory address range. |
| **getdomainname** | Gets the name of the current domain. |
| **getgroups** | Gets the concurrent group set of the current process. |
| **gethostname** | Gets the name of the local host. |
| **getpeername** | Gets the name of the peer socket. |
| **getrlimit** | Controls maximum system resource consumption. |
| **getrusage** | Displays information about resource use. |
| **getsockname** | Gets the socket name. |
| **getsockopt** | Gets options on sockets. |
| **gettimer** | Gets and sets the current value for the specified system-wide timer. |
| **resabs** | Manipulates the expiration time of interval timers. |
| **resinc** | Manipulates the expiration time of interval timers. |
| **restimer** | Gets and sets the current value for the specified system-wide timer. |
| **semctl** | Controls semaphore operations. |
| **semop** | Performs semaphore operations. |
| **setdomainname** | Sets the name of the current domain. |
| **setgroups** | Sets the concurrent group set of the current process. |
| **sethostname** | Sets the name of the current host. |
| **setrlimit** | Controls maximum system resource consumption. |
| **settimer** | Gets and sets the current value for the specified systemwide timer. |
| **shmat** | Attaches a shared memory segment or a mapped file to the current process. |
| **shmctl** | Controls shared memory operations. |
| **shmdt** | Detaches a shared memory segment. |
| **shmget** | Gets shared memory segments. |
| **sigaction** | Specifies the action to take upon delivery of a signal. |
| **sigprocmask** | Sets the current signal mask. |

| Item | Description |
| --- | --- |
| **sigstack** | Sets and gets signal stack context. |
| **sigsuspend** | Atomically changes the set of blocked signals and waits for a signal. |
| **sysconfig** | Provides a service for controlling system/kernel configuration. |
| **sys_parm** | Provides a service for examining or setting kernel run-time tunable parameters. |
| **times** | Displays information about resource use. |
| **uname** | Gets the name of the current system. |
| **unamex** | Gets the name of the current system. |
| **usrinfo** | Gets and sets user information about the owner of the current process. |
| **utimes** | Sets file access and modification times. |

# Virtual File Systems

The virtual file system (VFS) interface, also known as the v-node interface, provides a bridge between the physical and logical file systems. The information that follows discusses the virtual file system interface, its data structures, and its header files, and explains how to configure a virtual file system.

There are two essential components in the file system:

| Item | Description |
| --- | --- |
| **Logical file system** | Provides support for the system call interface. |
| **Physical file system** | Manages permanent storage of data. |

The interface between the physical and logical file systems is the *virtual file system interface*. This interface allows support for multiple concurrent instances of physical file systems, each of which is called a file system implementation. The file system implementation can support storing the file data in the local node or at a remote node. For more information on the virtual filesystem interface, see “Understanding the Virtual File System Interface” on page 35.

The virtual file system interface is usually referred to as the *v-node* interface. The v-node structure is the key element in communication between the virtual file system and the layers that call it. For more information on v-nodes, see “Understanding Virtual Nodes (V-nodes)” on page 34.

Both the virtual and logical file systems exist across all of this operating system family's platforms.

Review the following information for more information:

- **mntctl** subroutine
- **mount** subroutine
- **sysconfig** subroutine
- **vmount.h** file
- **gfsadd** kernel service
- **gfsdel** kernel service

**Related concepts**

Logical File System Kernel Services

The Logical File System services (also known as the **fp_**services) allow processes running in kernel mode to open and manipulate files in the same way that user-mode processes do. Data access limitations make it unreasonable to accomplish these tasks with system calls, so a subset of the file system calls has been provided with an alternate kernel-only interface.

Understanding Protection Domains

There are two protection domains in the operating system: the *user protection domain* and the *kernel mode protection domain*.

**Related information**

List of Virtual File System Operations

# Logical File System Overview

The *logical file system* is the level of the file system at which users can request file operations by system call. This level of the file system provides the kernel with a consistent view of what might be multiple physical file systems and multiple file system implementations. As far as the logical file system is concerned, file system types, whether local, remote, or strictly logical, and regardless of implementation, are indistinguishable.

A consistent view of file system implementations is made possible by the virtual file system abstraction. This abstraction specifies the set of file system operations that an implementation must include in order to carry out logical file system requests. Physical file systems can differ in how they implement these predefined operations, but they must present a uniform interface to the logical file system. A list of file system operators can be found at “Requirements for a File System Implementation” on page 35. For more information on the virual file system, see “Virtual File System Overview” on page 34.

Each set of predefined operations implemented constitutes a virtual file system. As such, a single physical file system can appear to the logical file system as one or more separate virtual file systems.

Virtual file system operations are available at the logical file system level through the *virtual file system switch*. This array contains one entry for each virtual file system, with each entry holding entry point addresses for separate operations. Each file system type has a set of entries in the virtual file system switch.

The logical file system and the virtual file system switch support other operating system file-system access semantics. This does not mean that only other operating system file systems can be supported. It does mean, however, that a file system implementation must be designed to fit into the logical file system model. Operations or information requested from a file system implementation need be performed only to the extent possible.

Logical file system can also refer to the tree of known path names in force while the system is running. A virtual file system that is mounted onto the logical file system tree itself becomes part of that tree. In fact, a single virtual file system can be mounted onto the logical file system tree at multiple points, so that nodes in the virtual subtree have multiple names. Multiple mount points allow maximum flexibility when constructing the logical file system view.

## Component Structure of the Logical File System

The logical file system is divided into system calls, logical file system file routines, and v-nodes components.

- System calls

  Implement services exported to users. System calls that carry out file system requests do the following:

  – Map the user's parameters to a file system object. This requires that the system call component use the v-node (virtual node) component to follow the object's path name. In addition, the system call must resolve a file descriptor or establish implicit (mapped) references using the open file component.

  – Verify that a requested operation is applicable to the type of the specified object.

  – Dispatch a request to the file system implementation to perform operations.

- Logical file system file routines

  Manage open file table entries and per-process file descriptors. An open file table entry records the authorization of a process's access to a file system object. A user can refer to an open file table entry through a file descriptor or by accessing the virtual memory to which the file was mapped. The logical

file system routines are those kernel services, such as **fp_ioctl** and **fp_select**, that begin with the prefix **fp_**.

- v-nodes

  Provide system calls with a mechanism for local name resolution. Local name resolution allows the logical file system to access multiple file system implementations through a uniform name space.

# Virtual File System Overview

The virtual file system is an abstraction of a physical file system implementation. It provides a consistent interface to multiple file systems, both local and remote. This consistent interface allows the user to view the directory tree on the running system as a single entity even when the tree is made up of a number of diverse file system types.

The interface also allows the logical file system code in the kernel to operate without regard to the type of file system being accessed. For more information on the logical file system, see "Logical File System Overview" on page 33.

A virtual file system can also be viewed as a subset of the logical file system tree, that part belonging to a single file system implementation. A virtual file system can be physical (the instantiation of a physical file system), remote, or strictly logical. In the latter case, for example, a virtual file system need not actually be a true file system or entail any underlying physical storage device.

A virtual file system mount point grafts a virtual file system subtree onto the logical file system tree. This mount point ties together a mounted-over v-node (virtual node) and the root of the virtual file system subtree. A mounted-over, or stub, v-node points to a virtual file system, and the mounted VFS points to the v-node it is mounted over.

## Understanding Virtual Nodes (V-nodes)

A *virtual node* (v-node) represents access to an object within a virtual file system. V-nodes are used only to translate a path name into a generic node (g-node).

For more information on g-nodes, see "Understanding Generic I-nodes (G-nodes)" on page 34.

A v-node is either created or used again for every reference made to a file by path name. When a user attempts to open or create a file, if the VFS containing the file already has a v-node representing that file, a use count in the v-node is incremented and the existing v-node is used. Otherwise, a new v-node is created.

Every path name known to the logical file system can be associated with, at most, one file system object. However, each file system object can have several names. Multiple names appear in the following cases:

- The object can appear in multiple virtual file systems. This can happen if the object (or an ancestor) is mounted in different virtual file systems using a local file-over-file or directory-over-directory mount.
- The object does not have a unique name within the virtual file system. (The file system implementation can provide synonyms. For example, the use of links causes files to have more than one name. However, opens of synonymous paths do not cause multiple v-nodes to be created.)

## Understanding Generic I-nodes (G-nodes)

A *generic i-node* (g-node) is the representation of an object in a file system implementation. There is a one-to-one correspondence between a g-node and an object in a file system implementation. Each g-node represents an object owned by the file system implementation.

Each file system implementation is responsible for allocating and destroying g-nodes. The g-node then serves as the interface between the logical file system and the file system implementation. Calls to the file system implementation serve as requests to perform an operation on a specific g-node.

A g-node is needed, in addition to the file system i-node, because some file system implementations may not include the concept of an i-node. Thus the g-node structure substitutes for whatever structure the file system implementation may have used to uniquely identify a file system object.

The logical file system relies on the file system implementation to provide valid data for the following fields in the g-node:

| Item | Description |
|---|---|
| **gn_type** | Identifies the type of object represented by the g-node. |
| **gn_ops** | Identifies the set of operations that can be performed on the object. |

## Understanding the Virtual File System Interface

Operations that can be performed upon a virtual file system and its underlying objects are divided into two categories.

Operations upon a file system implementation as a whole (not requiring the existence of an underlying file system object) are called **vfs** operations. Operations upon the underlying file system objects are called v-node (virtual node) operations. Before writing specific virtual file system operations, it is important to note the requirements for a file system implementation.

### *Requirements for a File System Implementation*

File system implementations differ in how they implement the predefined operations.

However, the logical file system expects that a file system implementation meets the following criteria:

- All **vfs** and v-node operations must supply a return value:
  - A return value of 0 indicates the operation was successful.
  - A nonzero return value is interpreted as a valid error number (taken from the **/usr/include/sys/errno.h** file) and returned through the system call interface to the application program.
- All **vfs** operations must exist for each file system type, but can return an error upon startup. The following are the necessary **vfs** operations:
  - **vfs_cntl**
  - **vfs_mount**
  - **vfs_root**
  - **vfs_statfs**
  - **vfs_sync**
  - **vfs_unmount**
  - **vfs_vget**
  - **vfs_quotactl**

For a complete list of file system operations, see List of Virtual File System Operations.

### *Important Data Structures for a File System Implementation*

There are two important data structures used to represent information about a virtual file system, the **vfs** structure and the v-node. Each virtual file system has a **vfs** structure in memory that describes its type, attributes, and position in the file tree hierarchy. Each file object within that virtual file system can be represented by a v-node.

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

| Item | Description |
|---|---|
| `vfs_flag` | Contains the state flags:<br>**VFS_DEVMOUNT**<br>Indicates whether the virtual file system has a physical mount structure underlying it.<br>**VFS_READONLY**<br>Indicates whether the virtual file system is mounted read-only. |

| Item | Description |
|---|---|
| vfs_type | Identifies the type of file system implementation. Possible values for this field are described in the **/usr/include/sys/vmount.h** file. |
| vfs_ops | Points to the set of operations for the specified file system type. |
| vfs_mntdover | Points to the mounted-over v-node. |
| vfs_data | Points to the file system implementation data. The interpretation of this field is left to the discretion of the file system implementation. For example, the field could be used to point to data in the kernel extension segment or as an offset to another segment. |
| vfs_mdata | Records the user arguments to the **mount** call that created this virtual file system. This field has a time stamp. The user arguments are retained to implement the **mntctl** call, which replaces the **/etc/mnttab** table. |

## Understanding Data Structures and Header Files for Virtual File Systems

This section lists the data structures used in implementing virtual file systems.

- The **vfs** structure contains information about a virtual file system as a single entity.
- The **vnode** structure contains information about a file system object in a virtual file system. There can be multiple v-nodes for a single file system object.
- The **gnode** structure contains information about a file system object in a physical file system. There is only a single g-node for a given file system object.
- The **gfs** structure contains information about a file system implementation. This is distinct from the **vfs** structure, which contains information about an instance of a virtual file system.

The header files contain the structure definitions for the key components of the virtual file system abstraction. Understanding the contents of these files and the relationships between them is essential to an understanding of virtual file systems. The following are the necessary header files:

- **sys/vfs.h**
- **sys/gfs.h**
- **sys/vnode.h**
- **sys/vmount.h**

## Configuring a Virtual File System

The kernel maintains a table of active file system types.

A file system implementation must be registered with the kernel before a request to mount a virtual file system (VFS) of that type can be honored. Two kernel services, **gfsadd** and **gfsdel**, are supplied for adding a file system type to the **gfs** file system table.

These are the steps that must be followed to get a file system configured.

1. A user-level routine must call the **sysconfig** subroutine requesting that the code for the virtual file system be loaded.
2. The user-level routine must then request, again by calling the **sysconfig** subroutine, that the virtual file system be configured. The name of a VFS-specific configuration routine must be specified.
3. The virtual file system-specific configuration routine calls the **gfsadd** kernel service to have the new file system added to the **gfs** table. The **gfs** table that the configuration routine passes to the **gfsadd** kernel service contains a pointer to an initialization routine. This routine is then called to do any further virtual file system-specific initialization.
4. The file system is now operational.

# Kernel Services

*Kernel services* are routines that provide the runtime kernel environment to programs executing in kernel mode. Kernel extensions call kernel services, which resemble library routines. In contrast, application programs call library routines.

Callers of kernel services execute in kernel mode. They therefore share with the kernel the responsibility for ensuring that system integrity is not compromised.

For a list of system calls that kernel extensions are allowed to use, see "System Calls Available to Kernel Extensions" on page 30.

Review the following information for associated command, subroutines, and kernel services descriptions:

- **setpri** subroutine
- **sysconfig** subroutine
- **pthread_cond_wait** subroutine
- **ar** command
- **ld** command
- **clrjmpx** kernel service
- **copyin** kernel service
- **copyinstr** kernel service
- **copyout** kernel service
- **creatp** kernel service
- **disable_lock** kernel service
- **e_sleep** kernel service
- **e_sleepl** kernel service
- **et_wait** kernel service
- **fubyte** kernel service
- **fuword** kernel service
- **getexcept** kernel service
- **i_disable** kernel service
- **i_enable** kernel service
- **i_init** kernel service
- **initp** kernel service
- **lockl** kernel service
- **longjmpx** kernel service
- **setjmpx** kernel service
- **setpinit** kernel service
- **sig_chk** kernel service
- **subyte** kernel service
- **suword** kernel service
- **uiomove** kernel service
- **unlockl** kernel service
- **ureadc** kernel service
- **uwritec** kernel service
- **uexadd** kernel service
- **uexdel** kernel service

- **xmalloc** kernel service
- **xmattach** kernel service
- **xmdetach** kernel service
- **xmemin** kernel service
- **xmemout** kernel service
- **uio** structure

**Related concepts**

Locking Kernel Services
The kernel services can be locked using any of the methods that are described in this overview.

Handling Signals While in a System Call
Signals can be generated asynchronously or synchronously with respect to the thread that receives the signal. An asynchronously generated signal is one that results from some action external to a thread. It is not directly related to the current instruction stream of that thread. Generally these are generated by other threads or by device drivers.

System Calls Available to Kernel Extensions
The system calls are grouped according to which subroutines call them.

# I/O Kernel Services

This overview lists the various I/O kernel services.

## Block I/O Kernel Services

This section lists the block I/O kernel services with a brief description.

| Item | Description |
|---|---|
| **iodone** | Performs block I/O completion processing. |
| **iowait** | Waits for block I/O completion. |
| **uphysio** | Performs character I/O for a block device using a **uio** structure. |

**Related concepts**

Block I/O Buffer Cache Kernel Services: Overview
The Block I/O Buffer Cache services are provided to support user access to device drivers through block I/O special files.

## Buffer Cache Kernel Services

This section lists the buffer cache kernel services with a brief description.

| Item | Description |
|---|---|
| **bawrite** | Writes the specified buffer's data without waiting for I/O to complete. |
| **bdwrite** | Releases the specified buffer after marking it for delayed write. |
| **bflush** | Flushes all write-behind blocks on the specified device from the buffer cache. |
| **binval** | Invalidates all of the specified device's blocks in the buffer cache. |
| **blkflush** | Flushes the specified block if it is in the buffer cache. |
| **bread** | Reads the specified block's data into a buffer. |
| **breada** | Reads in the specified block and then starts I/O on the read-ahead block. |
| **brelse** | Frees the specified buffer. |
| **bwrite** | Writes the specified buffer's data. |

| Item | Description |
|---|---|
| **clrbuf** | Sets the memory for the specified buffer structure's buffer to all zeros. |
| **getblk** | Assigns a buffer to the specified block. |
| **geteblk** | Allocates a free buffer. |
| **geterror** | Determines the completion status of the buffer. |
| **purblk** | Purges the specified block from the buffer cache. |

For information on how to manage the buffer cache with the Buffer Cache kernel services, see “Block I/O Buffer Cache Kernel Services: Overview” on page 49.

## Character I/O Kernel Services

This section lists the character I/O kernel services with description.

| Item | Description |
|---|---|
| **getc** | Retrieves a character from a character list. |
| **getcb** | Removes the first buffer from a character list and returns the address of the removed buffer. |
| **getcbp** | Retrieves multiple characters from a character buffer and places them at a designated address. |
| **getcf** | Retrieves a free character buffer. |
| **getcx** | Returns the character at the end of a designated list. |
| **pincf** | Manages the list of free character buffers. |
| **putc** | Places a character at the end of a character list. |
| **putcb** | Places a character buffer at the end of a character list. |
| **putcbp** | Places several characters at the end of a character list. |
| **putcf** | Frees a specified buffer. |
| **putcfl** | Frees the specified list of buffers. |
| **putcx** | Places a character on a character list. |
| **waitcfree** | Checks the availability of a free character buffer. |

## Interrupt Management Kernel Services

The operating system provides the following set of kernel services for managing interrupts.

See Understanding Interrupts for a description of these services:

| Item | Description |
|---|---|
| **i_init** | Defines an interrupt handler. |
| **i_eoi** | Issues an End of Interrupt (EOI) for a given handler. |
| **i_clear** | Removes an interrupt handler from the system. |
| **i_sched** | Schedules off-level processing. |
| **i_mask** | Disables an interrupt level. |
| **i_unmask** | Enables an interrupt level. |
| **i_disable** | Disables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a less-favored interrupt priority. |

| Item | Description |
|---|---|
| **i_enable** | Enables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a more-favored interrupt priority. |

## Memory Buffer (mbuf) Kernel Services

The Memory Buffer (mbuf) kernel services provide functions to obtain, release, and manipulate memory buffers, or **mbufs**.

These **mbuf** services provide the means to easily work with the **mbuf** data structure, which is defined in the **/usr/include/sys/mbuf.h** file. Data can be stored directly in an **mbuf**'s data portion or in an attached external cluster. **Mbufs** can also be chained together by using the `m_next` field in the **mbuf** structure. This is particularly useful for communications protocols that need to add and remove protocol headers.

The Memory Buffer (**mbuf**) kernel services are:

| Item | Description |
|---|---|
| **m_adj** | Adjusts the size of an **mbuf** chain. |
| **m_clattach** | Allocates an **mbuf** structure and attaches an external cluster. |
| **m_cat** | Appends one **mbuf** chain to the end of another. |
| **m_clgetm** | Allocates and attaches an external buffer. |
| **m_collapse** | Guarantees that an **mbuf** chain contains no more than a given number of **mbuf** structures. |
| **m_copydata** | Copies data from an **mbuf** chain to a specified buffer. |
| **m_copym** | Creates a copy of all or part of a list of **mbuf** structures. |
| **m_dereg** | Deregisters expected **mbuf** structure usage. |
| **m_free** | Frees an **mbuf** structure and any associated external storage area. |
| **m_freem** | Frees an entire **mbuf** chain. |
| **m_get** | Allocates a memory buffer from the **mbuf** pool. |
| **m_getclr** | Allocates and zeros a memory buffer from the **mbuf** pool. |
| **m_getclustm** | Allocates an **mbuf** structure from the **mbuf** buffer pool and attaches a cluster of the specified size. |
| **m_gethdr** | Allocates a header memory buffer from the **mbuf** pool. |
| **m_pullup** | Adjusts an **mbuf** chain so that a given number of bytes is in contiguous memory in the data area of the head **mbuf** structure. |
| **m_reg** | Registers expected **mbuf** usage. |

In addition to the **mbuf** kernel services, the following macros are available for use with **mbufs**:

| Item | Description |
|---|---|
| **m_clget** | Allocates a page-sized **mbuf** structure cluster. |
| **m_copy** | Creates a copy of all or part of a list of **mbuf** structures. |
| **m_getclust** | Allocates an **mbuf** structure from the **mbuf** buffer pool and attaches a page-sized cluster. |
| **M_HASCL** | Determines if an **mbuf** structure has an attached cluster. |
| **DTOM** | Converts an address anywhere within an **mbuf** structure to the head of that **mbuf** structure. |

| Item | Description |
|---|---|
| **MTOCL** | Converts a pointer to an **mbuf** structure to a pointer to the head of an attached cluster. |
| **MTOD** | Converts a pointer to an **mbuf** structure to a pointer to the data stored in that **mbuf** structure. |
| **M_XMEMD** | Returns the address of an **mbuf** cross-memory descriptor. |

## DMA Management Kernel Services

The operating system kernel provides several services for managing direct memory access (DMA) channels and performing DMA operations.

Understanding DMA Transfers provides additional kernel services information.

The services provided are:

| Item | Description |
|---|---|
| **d_align** | Provides needed information to align a buffer with a processor cache line. |
| **d_cflush** | Flushes the processor and I/O controller (IOCC) data caches when using the long term **DMA_WRITE_ONLY** mapping of DMA buffers approach to the bus device DMA. |
| `d_map_clear` | Deallocates resources previously allocated on a d_map_init call. |
| `d_map_disable` | Disables DMA for the specified handle. |
| `d_map_enable` | Enables DMA for the specified handle. |
| **d_map_init** | Allocates and initializes resources for performing DMA with PCI and ISA devices. |
| `d_map_list` | Performs platform-specific DMA mapping for a list of virtual addresses. |
| `d_map_page` | Performs platform-specific DMA mapping for a single page. |
| `d_map_slave` | Accepts a list of virtual addresses and sizes and sets up the slave DMA controller. |
| **d_roundup** | Rounds the value length up to a given number of cache lines. |
| `d_unmap_list` | Deallocates resources previously allocated on a `d_map_list` call. |
| `d_unmap_page` | Deallocates resources previously allocated on a `d_map_page` call. |
| `d_unmap_slave` | Deallocates resources previously allocated on a `d_map_slave` call. |

## Enhanced I/O Error Handling Kernel Services

Enhanced I/O Error Handling (EEH) kernel services is an error recovery strategy for errors that occur during I/O operations on a Peripheral Component Interconnect (PCI), Peripheral Component Interconnect Extended (PCI-X), or PCI Express (PCIe) bus.

Bridges and switches, PCI-to-PCI or PCI-X-to-PCI-X PCIe switches, that enable each slot to be on its own bus provide a form of electrical and logical isolation of slots. The smallest PCI unit (device or function) that can be assigned to a logical partition (LPAR) is called a *Partitionable Endpoint (PE)*.

For example, if each port of a 4-port Ethernet adapter can be assigned to a different LPAR, and each port is a PCI function, then each port would be a PE. In this case, granularity of a PE is a PCI function. A PE is not necessarily the same as a PCIe Endpoint, and the two should not be confused. A bridge or switch above the PE where the EEH state is maintained forms a PE domain. This bridge or switch is called the *Top of PE domain*. EEH recovery is performed according to the PE domain and is carried out by the Top of PE domain as directed by the software (operating system and device drivers).

Several PCI functions in one or more adapters that belong to the same EEH recovery domain are called a *shared EEH domain*. This has been typically limited to a multifunction adapter, in which the functions on

the adapter are recovered together. Because a shared EEH domain supports any number of PCI functions to be included in it, including the functions on different adapters, its function is more general than a multifunction adapter. For present purposes, the multifunction model are referred to as the *shared EEH model*.

In the LPAR environment, a PE domain is the same as a shared EEH domain and includes all PCI functions in the PE domain. In other words, if multiple functions belong to a shared EEH domain, they cannot be individual PEs because the EEH recovery can only affect the LPAR to which the PE belongs.

The types of adapters supported in the slot created by a PE domain are:

- Single-function adapter with or without a bridge or switch on the adapter.
- Multifunction adapter without a bridge or switch on the adapter
- Multifunction adapter with a bridge or switch on the adapter.

The bridges can be of different types, such as PCI-to-PCI or PCIX-to-PCI, and so on. A switch is a PCIe switch, which is logically a collection of bridges. The exact type of the bridge or switch is not important to this programming model. These details are handled by the hardware and firmware. A bridged single-function adapter is treated like a bridged multifunction adapter for the purposes of EEH programming model.

The device drivers for all these types of adapters use the same EEH kernel services to drive the error recovery except for the registration service. A nonbridged single-function adapter calls the `eeh_init()` registration service function. Adapters in a shared EEH domain call the `eeh_init_multifunc()` function. This includes any bridged or nonbridged multifunction adapters and bridged single-function adapters. Although a nonbridged single-function adapter typically calls `eeh_init()`, it can choose the shared EEH model and call `eeh_init_multifunc()` instead. Regardless of the number of functions and bridges, the device drivers should always use the shared EEH model and call `eeh_init_multifunc()`. The PCIe device drivers are required to use the shared EEH model. Although the same services are used by the single and shared EEH adapter drivers, the error recovery models are different. In addition, any time there are intermediate bridges between the Top of PE domain and the PCI functions in the PE domain, those bridges have to be recovered as well. For example, a multifunction bridged adapter requires that the bridge on the adapter also be recovered.

The error recovery is performed by resetting the PCI bus between the Top of PE domain and the PE under it, and then reconfiguring any intermediate bridges. The basic steps in error detection and recovery are as follows:

- An adapter driver suspects an error on the card when it receives some invalid values from one or more locations in its I/O or memory spaces.
- The driver then confirms the existence of the error by calling EEH kernel services. After the error state is confirmed, the slot is declared frozen.
- After the slot is frozen, all further activities to the card are suspended until the error is recovered. For example, new read/write requests are blocked or failed.
- The driver attempts to recover the slot by toggling the reset line. After three attempts to recover, the driver declares the slot unusable (or dead). If the slot is reset successfully, normal operations resume.

The key difference in the single-function and shared EEH models is that in the shared EEH model, there is a need for coordination among different driver instances controlling the same PE domain. For example, a PE domain can include a physical device on a single slot. The driver instances controlling each function on the device require coordination. In addition, there are steps in the recovery process that need to be carried out only once. So among all registered drivers in a shared EEH domain, one is chosen to be the *master*. The drivers follow a state machine. The EEH kernel services are implemented so that they present an EEH recovery state machine to the device drivers. It is the master driver's responsibility to drive the state machine. The section titled Shared EEH Programming Model, which follows, contains more details on how a master driver is determined. Many details are hidden from the device drivers for simplicity. Because the shared EEH model is more flexible and extensible, it is recommended for the new device drivers.

In the single-function model, the device drivers are responsible for driving their own error recovery. In other words, they are responsible for implementing their own state machine. Every time EEH recovery is extended in some way at the hardware or firmware level, there is probably a code and testing impact on the single-function implementations. As previously described, a single-function adapter should still use the shared EEH model. In that case, all the messages from the EEH kernel services are sent to just one driver instance.

### *Shared EEH Programming Model*

For the shared EEH programming model, the EEH kernel services present the following state machine to the drivers:

1. A slot starts out in the NORMAL state.
2. When an EEH event happens, the driver receives all F's from an MMIO load. Because all F's might be a legal value for a driver, the driver must call `eeh_read_slot_state()` to confirm the event.
3. If eeh_read_slot_state() finds the slot to be frozen, it broadcasts an EEH_DD_SUSPEND message to all registered drivers, and the slot state moves to SUSPEND. The kernel messages like this one are broadcast by invoking the callback routine sequentially. The messages are broadcast at INTIODONE priority.
4. When the drivers receive the EEH_DD_SUSPEND message, they can do one of the following:
   a. Gather some debug data from the adapter and proceed to reset the slot.

      Gathering the debug data is really an optional step in the recovery process, where a driver can choose to read certain registers on the adapter in an attempt to understand what caused the EEH event in the first place.

      To gather the debug data, the drivers must enable PIO to the adapter. PIO is frozen when an EEH event occurs. To enable PIO:

      i) The master driver must call `eeh_enable_pio()`. The master driver is picked by the EEH kernel services. It has the EEH_MASTER flag set on the callback routine and is the last driver called in the callback chain. This ensures that all other drivers in the shared EEH domain have finished the last step of the recovery and that the master driver can now proceed to the next step (such as enabling PIO).

      When `eeh_enable_pio()` is called, an EEH_DD_DEBUG message is sent to the drivers indicating that PIO is enabled, and the slot state moves to DEBUG.

      ii) The drivers then gather the data.

      eeh_enable_pio() can be called multiple times. Each time it is called, another EEH_DD_DEBUG message is broadcast.

      iii) When the drivers receive EEH_DD_SUSPEND or EEH_DD_DEBUG messages, they call eeh_slot_error() to create an AIX® error log entry with hardware debug data. This step is required to figure out the reason for the EEH event.

      iv) The master driver must call eeh_reset_slot() to reset the slot. Only one driver calls reset because it is not necessary to reset the slot multiple times.

   b. Proceed directly to reset the slot.
5. The reset line on the PCI bus is toggled with 100 ms delay between activate and deactivate to reset the slot. The delay is hidden from the device drivers and is enforced by the eeh_reset_slot() kernel service internally. The slot internally moves through the ACTIVATE and the DEACTIVATE states.
6. If there are any intermediate bridges present (such as a bridge on the adapter), at the end of a successful reset, EEH kernel services configures the bridge using eeh_configure_bridge() service. Kernel services also enforces a certain amount of delay between the deactivation of the reset line and the configuration of bridge.

   The device drivers do not need to call eeh_configure_bridge() directly.
7. If everything goes well, the EEH_DD_RESUME message is sent to the drivers indicating that the slot recovery is complete.

8. At this point, most drivers would have to reinitialize their adapters before starting normal operations again. Reinitialization typically requires a partial restore of the config space (such as the BARs and Cache Line). Determining the config space registers to be restore depends on the device.

   **Note:** This is the usual recovery sequence. If any of the services fail, the EEH_DD_DEAD message is broadcast asking the drivers to mark their adapters unavailable (for example, the drivers might have to perform some cleanup work and mark their internal states appropriately). The master driver must call eeh_slot_error() to create an AIX® error log and mark the adapter permanently unavailable.

There are two special scenarios that a driver developer needs to be aware of:

1. If a driver receives either an EEH_DD_SUSPEND or an EEH_DD_DEAD message, it can return an EEH_BUSY return code from its callback routine instead of an EEH_SUCC return code. If EEH kernel services receives an EEH_BUSY message, EEH kernel services waits for some time and then calls the same driver again. This process continues until EEH kernel services receive a different return code. This process is repeated because some drivers need more time to cleanup before recovery can continue. Cleanup would include such activities like killing a kproc or notifying a user level app.
2. If `eeh_enable_dma()` and `eeh_enable_pio()` cannot succeed due to the platform state restrictions, the service returns an EEH_FAIL return code followed by an EEH_DD_DEAD message unless you take action. To avoid receiving an EEH_FAIL return code, the driver must supply an *EEH_ENABLE_NO_SUPPORT_RC* flag when `eeh_init_multifunc()` kernel services is initiated. If an *EEH_ENABLE_NO_SUPPORT_RC* flag is supplied, `eeh_enable_pio()` and `eeh_enable_dma()` return the EEH_NO_SUPPORT return code that indicates to the drivers that they cannot collect debug data but can continue with the next step in recovery. For more information, see eeh_read_slot_state.

The EEH kernel services that you can use are listed in the following table:

**Note:** eeh_init() and eeh_init_multifunc() are the only exported kernel services. All other kernel services are called using function pointers in the eeh_handle kernel service.

| Kernel Service | Single Function | Shared EEH | Process Environment | Interrupt Environment |
|---|---|---|---|---|
| eeh_init | Y | N | Y | N |
| eeh_init_multifunc | N | Y | Y | N |
| eeh_clear | Y | Y | Y | N |
| eeh_read_slot_state | Y | Y | Y | Y |
| eeh_enable_pio | Y | Y | Y | Y |
| eeh_enable_dma | Y | Y | Y | Y |
| eeh_enable_slot | Y | N | Y | Y |
| eeh_disable_slot | Y | N | Y | Y |
| eeh_reset_slot | Y | Y | Y | Y |
| eeh_slot_error | Y | Y | Y | Y |
| eeh_broadcast | N | Y | Y | Y |

### *Callback Routine*

This section provides the definition of the `(*callback_ptr)()` function prototype.

```
long eeh_callback(
                unsigned long long cmd, /* EEH messages */
                void *arg,              /* Pointer to dd defined argument */
                unsigned long flag)     /* DD defined flag */
```

- *cmd* – contains a kernel and driver message

- *arg* – is a cookie to a target device driver that is usually a pointer to the adapter structure
- *flag* argument can be either just EEH_MASTER or EEH_MASTER ORed with EEH_DD_PIO_ENABLED

**EEH_MASTER**
Indicates that the target device driver is the EEH_MASTER.

**EEH_DD_PIO_ENABLED**
Set only with the EEH_DD_DEBUG message to indicate that the PIO is enabled.

When `eeh_init_multifunc()` is called, the callback routines are registered. When `eeh_clear()` is called the callback routines are unregistered. The callback routines are necessary for EEH kernel services recovery. They coordinate shared EEH domain driver instances. For more information on how this coordination is done, see "Enhanced I/O Error Handling Kernel Services" on page 41.

The shared EEH domain drivers are expected to handle the following EEH kernel services messages:

**EEH_DD_SUSPEND**
Notifies all the device drivers on a slot that an EEH kernel services event occurred. The slot is either frozen or temporarily unavailable. Because an EEH kernel services event occurred, the device drivers suspend operations. Then, the EEH_MASTER driver either enables PIO or resets the slot.

**EEH_DD_DEBUG**
Notifies all drivers on a slot that they can now gather debug data from the devices. The device drivers log errors by calling the `eeh_slot_error()` function and passing in the gathered debug data. This message is sent when the EEH_MASTER calls the `eeh_enable_pio()` function. On the callback routine, the flag argument is set to EEH_DD_PIO_ENABLED.

**EEH_DD_DEAD**
Notifies all drivers on a slot that the slot reached an unrecoverable state and the slot is no longer usable. This message is sent anytime EEH kernel services fail because of hardware or firmware problems. This message is also broadcast when a driver calls the `eeh_slot_error()` function with the flag set to EEH_RESET_PERM. The device drivers usually perform necessary cleanup and mark the adapter as permanently unavailable.

**EEH_DD_RESUME**
Notifies all drivers on a slot that the EEH kernel services event was recovered successfully and that the callback routines can now resume normal operation. This message is sent at the end of a successful toggle of reset line and optional bridge configuration (for example, the bridge on the adapter). The device drivers must usually re-initialize their adapters before normal operation can begin again.

The device drivers define their own messages based on the contents of the `sys/eeh.h` file.

The *eeh_callback()* functions are scheduled to start sequentially at INTIODONE priority. They are not started in any specific order. For more information, see eeh_broadcast.

### *EEH user message broadcast*

This topic describes the process of coordinating multiple functions in a device by using the kernel services. The user-state and user messages are defined by the caller of these services and the user messages are required to handle all serialization when using these services.

## User-state

The Extending Enhanced I/O Error Handling (EEH) framework maintains a user-state that is managed by the participating device drivers in the EEH-shared domain. User-state is defined as a 32-bit field per domain. User-state is initialized to 0 when the domain is first created on the first **eeh_init_multifunc()** service call for that domain. The bits in the user-state field have no predefined meaning. Their meaning is determined by the participating device drivers. The field might be treated as a **bit** field or as an **unsigned long** value. A START bit field message can be defined as 0x010 and the STOP bit field message can be defined as 0x001. An unsigned long value can set START bit field message as 7225 and the STOP bit field message as 7224. The device drivers define and coordinate these states.

## User messages

The device drivers define up to 32 user messages. Each message consumes one bit in the 32-bit message field. Only one message is broadcast at a time. All device drivers, including the **EEH_INITIATOR** driver are called for the message before moving on to the next message. The messages are sent from the leftmost bit to the rightmost bit without any priority. Before starting to broadcast the message, the bit for that message is cleared. This allows the same message to be queued again. If you queue a message more than once before the message is broadcast, it results in only one broadcast for the message. The messages are sent at an INTIODONE priority.

The messages are initialized to 0 when the domain is first created. The bits in the user message field have no predefined meaning. Their meanings are determined by the participating device drivers. The field might be treated only as a bit-field, where, each bit corresponds to a message.

## User message session

An user message session is the time between a function call to the **eeh_set_and_broadcast_user_state()** service, making it the **EEH_INITIATOR** function, to the time it calls the **eeh_clear_user_session()** service. During a single session, only the **EEH_INITIATOR** driver is allowed to call the **eeh_set_and_broadcast_user_state()** service. The calls to the **eeh_set_and_broadcast_user_state()** service by the **non-EEH_INITIATOR** driver return a kerrno based on the EBUSY. The session ends when the **EEH_INITIATOR** driver calls the **eeh_clear_user_session()** service, at which point the next function to call the **eeh_set_and_broadcast_user_state()** service starts the next session and becomes the new **EEH_INITIATOR**. All the **EEH_INITIATOR** driver are required to call the **eeh_clear_user_session()** service to complete their sessions.

The sessions also ends when an EEH event occurs, which clears any pending messages and clears the identity of the **EEH_INITIATOR** function.

When the **eeh_callback** service is called for a broadcast, the **flags** field in the callback message includes an **EEH_INITIATOR** flag for the **EEH_INITIATOR** driver , which is the last to receive the message.

It is highly recommended that the **EEH_INITIATOR** driver calls the **eeh_clear_user_session()** service before calling the **EEH_CLEAR** service. If the **EEH_INITIATIOR** function calls to **EEH_CLEAR** driver without ending the session, the **EEH_CLEAR** driver ends the session.

**Note:** The **EEH_INITIATOR** driver is defined in the **eeh.h** file.

## Managing the user-state and messages using kernel services

The following new services are created to manage the user messaging framework. These services and their flags are defined in the **eeh.h** file.

*eeh_set_and_broadcast_user_state*

## Purpose

Sets a user-state and broadcasts a user message.

## Parameter

| Item | Description |
|---|---|
| ***Handle*** | Extending Enhanced I/O Error Handling (EEH) handle acquired from the **eeh_init_multifunc()** service. |
| ***Flags*** | Additional flags. Currently, the value must be 0. |
| ***State*** | User-defined state value. |
| ***Message*** | User-defined message value. |

## Description

This service allows a device driver to change a user-defined state and to send a message to all other device drivers in the same EEH-shared domain. The caller of this service is marked **EEH_INITIATOR** for the message and is called with the previous message. The messages are broadcast at an INTIODONE priority. The EEH callback handler that is registered during the registration of the **eeh_init_multifunc()** service time is used to callback each device driver with the message.

## Execution Environment

The **eeh_set_and_broadcast_user_state()** service either process or interrupt. This function can be called from the live dump, the eeh callback, and other process level entry points. The **eeh_set_and_broadcast_user_state()** service will not work in the system dump context.

## Return Values

| Item | Description |
|---|---|
| **0** | Success |
| **Kerrno with EBUSY** | Session is currently active with another initiator |
| **Kerrno** | Error |

*eeh_get_user_state*

## Purpose

Reads a user-state.

## Syntax

```
kerrno_t
eeh_get_user_state(eeh_handle_t handle, ulong flags, uint *state,
        ulong* status)
```

## Parameter

| Item | Description |
|---|---|
| ***Handle*** | Extending Enhanced I/O Error Handling (EEH) handle acquired from the **eeh_init_multifunc()** service. |
| ***Flags*** | Additional flags. Currently, the value must be 0. |
| ***State*** | Pointer to where the user-state is returned. |
| ***Message*** | Bitmask information about the current session. |

## Description

This service provides a method for a device driver to read the user state. This service does not send any messages.

## Execution Environment

The **eeh_get_user_state()** service either process or interrupt. This service can be called from the live dump, the eeh callback, and other process level entry points.

## Return Values

| Item | Description |
|---|---|
| **0** | Success |
| **Kerrno** | Error |

## Status Flags

The status parameter currently defines the following flags:

| Flags | Description | Value |
|---|---|---|
| SESSION_ACTIVE | User Message Session is currently active. | 0x1 |
| IS_INITIATOR | User is EEH_INITIATOR | 0x2 |
| BRDCAST_ACTIVE | User Message is currently being broadcast. | 0x3 |

The flags are returned on success and also by the kerrno messages that are based on the EBUSY flag

*eeh_clear_user_session*

## Purpose

Clears a user-state and pending user messages, and ends a user message session.

## Syntax

```
kerrno_t
eeh_clear_user_session(eeh_handle_t handle, ulong flags)
```

## Parameter

| Item | Description |
|---|---|
| ***Handle*** | Extending Enhanced I/O Error Handling (EEH) handle acquired from the **eeh_init_multifunc()** service. |
| ***Flags*** | Additional flags for the **eeh_clear_user_session()** service. |

## Description

This service is called if a device driver wants to reset the state to 0 and clear any pending user messages. This service also clears the identity of the **EEH_INITIATOR** driver. The drivers that are calling the **eeh_clear_user_session()** service and that are not defined as the **EEH_INITIATOR** driver receive a return value of a kerrno, which is based on the EBUSY state, with the exception of when the device driver specifies the **EEH_FORCE_CLEAR** driver. This ends the current session. The **EEH_FORCE_CLEAR** driver is defined in the **eeh.h** file.

## Execution Environment

The **eeh_clear_user_session()** service either process or interrupt. This function can be called from the live dump, the eeh callback, and other process level entry points.

### Return Values

| Item | Description |
|---|---|
| **0** | Success |
| **Kerrno with EBUSY** | Session is currently active with another initiator |
| **Kerrno** | Error |

## Block I/O Buffer Cache Kernel Services: Overview

The Block I/O Buffer Cache services are provided to support user access to device drivers through block I/O special files.

This access is required by the operating system file system for mounts and other limited activity, as well as for compatibility services required when other file systems are installed on these kinds of systems. These services are not used by the operating system's JFS (journal file system), NFS (Network File System), or CDRFS (CD-ROM file system) when processing standard file I/O data. Instead they use the virtual memory manager and pager to manage the system's memory pages as a buffer cache.

For compatibility support of other file systems and block special file support, the buffer cache services serve two important purposes:

- They ensure that multiple processes accessing the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block.
- They increase the efficiency of the system by keeping in-memory copies of blocks that are frequently accessed.

The Buffer Cache services use the **buf** structure or buffer header as their main data-tracking mechanism. Each buffer header contains a pair of pointers that maintains a doubly-linked list of buffers associated with a particular block device. An additional pair of pointers maintain a doubly-linked list of blocks available for use again on another operation. Buffers that have I/O in progress or that are busy for other purposes do not appear in this available list.

Kernel buffers are discussed in more detail in Introduction to Kernel Buffers.

See “Block I/O Kernel Services” on page 38 for a list of these services.

**Related reference**

Block I/O Kernel Services
This section lists the block I/O kernel services with a brief description.

## Managing the Buffer Cache

Fourteen kernel services provide management of this block I/O buffer cache mechanism.

The **getblk** kernel service allocates a buffer header and a free buffer from the buffer pool. Given a device and block number, the **getblk** and **bread** kernel services both return a pointer to a buffer header for the block. But the **bread** service is guaranteed to return a buffer actually containing a current data for the block. In contrast, the **getblk** service returns a buffer that contains the data in the block only if it is already in memory.

In either case, the buffer and the corresponding device block are made busy. Other processes attempting to access the buffer must wait until it becomes free. The **getblk** service is used when:

- A block is about to be rewritten totally.
- Its previous contents are not useful.
- No other processes should be allowed to access it until the new data has been placed into it.

The **breada** kernel service is used to perform read-ahead I/O and is similar to the **bread** service except that an additional parameter specifies the number of the block on the same device to be read asynchronously after the requested block is available. The **brelse** kernel service makes the specified buffer available again to other processes.

### Using the Buffer Cache write Services

There are three slightly different write routines. All of them take a buffer pointer as a parameter and all logically release the buffer by placing it on the free list.

The **bwrite** service puts the buffer on the appropriate device queue by calling the device's strategy routine. The **bwrite** service then waits for I/O completion and sets the caller's error flag, if required. This service is used when the caller wants to be sure that I/O takes place synchronously, so that any errors can be handled immediately.

The **bawrite** service is an asynchronous version of the **bwrite** service and does not wait for I/O completion. This service is normally used when the overlap of processing and device I/O activity is desired.

The **bdwrite** service does not start any I/O operations, but marks the buffer as a delayed write and releases it to the free list. Later, when the buffer is obtained from the free list and found to contain data from some other block, the data is written out to the correct device before the buffer is used. The **bdwrite** service is used when it is undetermined if the write is needed immediately.

For example, the **bdwrite** service is called when the last byte of the write operation associated with a block special file falls short of the end of a block. The **bdwrite** service is called on the assumption that another write will soon occur that will use the same block again. On the other hand, as the end of a block is passed, the **bawrite** service is called, because it is assumed the block will not be accessed again soon. Therefore, the I/O processing can be started as soon as possible.

Note that the **getblk** and **bread** services dedicated the specified block to the caller while making other processes wait, whereas the **brelse**, **bwrite**, **bawrite**, or **bdwrite** services must eventually be called to free the block for use by other processes.

## Understanding Interrupts

Each hardware interrupt has an interrupt level, trigger, and interrupt priority.

The following sections describe various interrupt components:

**Related concepts**

System Calls
A system call is a routine that allows a user application to request actions that require special privileges. Adding system calls is one of several ways to extend the functions provided by the kernel.

### Interrupt Level

The interrupt level defines the source of the interrupt and is often referred to as the *interrupt source*.

There are basically two types of interrupt levels: system and bus. The bus interrupts are generated by the devices on the buses (such as PCI, ISA, VDEVICE, and PCI-E). Examples of system interrupts are the timer and Environmental and Power Off Warning (EPOW).

The interrupt level of a bus or system interrupt is one of the resources managed by the respective configuration methods.

### Interrupt Trigger

There are two types of trigger mechanisms, level-triggered interrupts and edge-triggered interrupts.

All ISA and VDEVICE interrupts are edge-triggered. PCI/PCIX and PCI-E buses define two types of interrupts, Level Signalled Interrupts (LSI) and Message Signalled Interrupts (MSI). LSIs are level-triggered, and MSIs are edge-triggered. PCI/PCIX device drivers in AIX must handle only level-triggered interrupts even though edge-triggered interrupts using MSIs are supported by PCIX. Similarly, even though PCI-E supports LSI interrupts, all PCI-E interrupts are MSIs only in AIX, and are therefore edge-triggered. Consequently, all PCI-E device drivers for AIX are required to handle only edge-triggered interrupts.

A key difference between the edge-triggered and level-triggered interrupts is interrupt sharing. Level-triggered interrupts can be shared. Edge-triggered interrupts cannot be shared. Because they cannot be shared, edge-triggered interrupt handlers should pass the INTR_EDGE flag on the `i_init()` kernel service.

Another difference between the edge-triggered and level-triggered interrupts is in issuing the End of Interrupt (EOI). For level-triggered interrupts, the AIX kernel issues the EOI. For ISA edge-triggered interrupts, the AIX kernel also issues EOI. However, for the VDEVICE and PCI-E edge-triggered interrupts, the device driver must issue the EOI before returning from the interrupt handler. The VDEVICE and PCI-E drivers can use the `i_eoi()` kernel service to accomplish this.

**Note:** During the processing of `i_eoi()`, there is a brief period of time in which a newly arrived interrupt could also be issued an EOI. Therefore, it is necessary to check for additional work between a call to `i_eoi()` and the return from the interrupt handler.

### Interrupt Priorities

The interrupt priority defines which of a set of pending interrupts is serviced first. INTMAX is the most favored interrupt priority and INTBASE is the least favored interrupt priority. The interrupt priorities for bus interrupts range from INTCLASS0 to INTCLASS3. The rest of the interrupt priorities are reserved for the base kernel. Interrupts that cannot be serviced within the time limits specified for bus interrupts qualify as off-level interrupts.

A device's interrupt priority is selected based on two criteria: its maximum interrupt *latency* requirements and the device driver's interrupt *execution time*. The interrupt latency requirement is the maximum time within which an interrupt must be serviced. (If it is not serviced in this time, some event is lost or performance is seriously degraded.) The interrupt execution time is the number of machine cycles required by the device driver to service the interrupt. Interrupts with a short interrupt latency time must have a short interrupt service time.

The general rule for interrupt service times is based on the following interrupt priority table:

| Priority | Service Time (machine cycles) |
|---|---|
| **INTCLASS0** | 200 cycles |
| **INTCLASS1** | 400 cycles |
| **INTCLASS2** | 600 cycles |
| **INTCLASS3** | 800 cycles |

The valid interrupt priorities are defined in the **/usr/include/sys/intr.h** file.

See “Interrupt Management Kernel Services” on page 39 for a list of these services.

## Understanding DMA Transfers

AIX DMA support deals with the issues of direct memory access by I/O devices to and from system memory.

The programming framework supports common I/O buses (such as PCI, industry standard architecture (ISA), and PCIe) and is easily extensible to additional bus types in the future. The framework supports 64-bit addressability, and also allows for mappings from 32-bit devices to 64-bit addresses to be hidden from the devices and their drivers.

### DMA Programming Model

This is the basic DMA programming model.

It is completely independent of:

- System hardware
- LPAR mode or non-LPAR mode

- 32-bit bus/devices or 64-bit bus or devices
- 32-bit kernel or 64-bit kernel

A device driver allocates and initializes DMA-related resources with the d_map_init service and the **d_map_init_ext** service and frees the resources with the d_map_clear service. Each time a DMA mapping needs to be established, the driver calls d_map_page or d_map_list service.

The d_map_page and d_map_list services map DMA buffers in the bus memory. In other words, given a set of DMA buffer addresses, a corresponding set of bus addresses is returned to the driver. The driver programs its device with the bus addresses and sets it up to start the DMA. When the DMA is complete:

- The device generates an interrupt that is handled by the driver.
- If no more DMA will be done to the buffers, the driver unmaps the DMA buffers with the d_unmap_page or d_unmap_list services.

## Data Structures

The **d_map_init** service and the **d_map_init_ext** service return a **d_handle_t** structure to the caller upon a successful completion. Only the **d_map_init** service and the **d_map_init_ext** service are the exported kernel services.

All other DMA kernel services are called through the function pointers in the d_handle_t structure (see the sys/dma.h system file).

```
d_handle {
               uint id; /* identifier for this device */
               uint flags; /* device capabilities */
#ifdef __64BIT_KERNEL
               /* pointer to d_map_page routine */
               int (*d_map_page)(d_handle_t,int,caddr_t, ulong *, struct xmem
*);
               /* pointer to d_unmap_page routine */
               void (*d_unmap_page)(d_handle_t, ulong *);
               /* pointer to d_map_list routine */
               int (*d_map_list)(d_handle_t, int, int, dio_t, dio_t);
               /* pointer to d_unmap_list routine */
               void (*d_unmap_list)(d_handle_t, dio_t);
               /* pointer to d_map_slave routine */
               int (*d_map_slave)(d_handle_t, int, int, dio_t, uint);
               /* pointer to d_unmap_slave routine */
               int (*d_unmap_slave)(d_handle_t);
               /* pointer to d_map_disable routine */
               int (*d_map_disable)(d_handle_t);
               /* pointer to d_map_enable routine */
               int (*d_map_enable)(d_handle_t);
               /* pointer to d_map_clear routine */
               void (*d_map_clear)(d_handle_t);
               /* pointer to d_sync_mem routine */
               int (*d_sync_mem)(d_handle_t, dio_t);
#else
               int (*d_map_page)(); /* pointer to d_map_page routine */
               void (*d_unmap_page)(); /* pointer to d_unmap_page routine */
               int (*d_map_list)(); /* pointer to d_map_list routine */
               void (*d_unmap_list)(); /* pointer to d_unmap_list routine */
               int (*d_map_slave)(); /* pointer to d_map_slave routine */
               int (*d_unmap_slave)(); /* pointer to d_unmap_slave routine */
               int (*d_map_disable)(); /* pointer to d_map_disable routine */
               int (*d_map_enable)(); /* pointer to d_map_enable routine */
               void (*d_map_clear)(); /* pointer to d_map_clear routine */
               int (*d_sync_mem)(); /* pointer to d_sync_mem routine */
#endif
               int bid; /* bus id passed to d_map_init or d_map_init_ext*/
               void *bus_sys_xlate_ptr; /* pointer to dma bus to system
               translation information */
               uint reserved1; /* padding */
               uint reserved2; /* padding */
               uint reserved3; /* padding */
};
```

The following dio and d_iovec structures are used to define the scatter and gather lists used by the d_map_list, d_unmap_list, and d_map_slave services (see the sys/dma.h system file).

```
struct dio {
               int32long64_t total_iovecs; /* total available iovec entries */
               int32long64_t used_iovecs; /* number of used iovecs */
               int32long64_t bytes_done; /* count of bytes processed */
               int32long64_t resid_iov; /* number of iovec that couldn't be */
                                        /* fully mapped due to NORES,DIOFULL*/
                                        /* vec =&dvec [resid_iov] */
               struct d_iovec *dvec; /* pointer to list of d_iovecs */
};

struct d_iovec {
               caddr_t iov_base; /* base memory address */
               int32long64_t iov_len; /* length of transfer for this area */
               struct xmem *xmp; /* cross memory pointer for this address*/
};
```

The following dio_64 and d_iovec_64 structures are used to define the scatter and gather lists used by the d_map_list and d_unmap_list services when the DMA_ENABLE_64 flag is set on the d_map_init call or the **d_map_init_ext** call. These are not used under the 64-bit kernel and kernel extension environment because the dio and d_iovec data structures are naturally 64-bit capable in that environment. (For more information, see the sys/dma.h system file.)

```
struct dio_64 {
               int total_iovecs; /* total available iovec entries */
               int used_iovecs; /* number of used iovecs */
               int bytes_done; /* count of bytes processed */
               int resid_iov; /* number of iovec that couldn't be */
                              /* fully mapped due to NORES,DIOFULL*/
                              /* vec = &dvec [resid_iov] */
               struct d_iovec_64 *dvec; /* pointer to list of d_iovecs */
};

struct d_iovec_64 {
               unsigned long long iov_base; /* base memory address */
               int iov_len; /* length of transfer for this area */
               struct xmem *xmp; /* cross memory pointer for this address*/
}
```

The following macros are provided in sys/dma.h for device drivers in order to call the DMA kernel services cleanly:

```
#define D_MAP_INIT(bid, flags, bus_flags, channel) \
                  d_map_init(bid, flags, bus_flags, channel)

#define D_MAP_INIT_EXT(dma_input, info_size, handle_ptr) \
                  d_map_init_ext(dma_input, info_size, handle_ptr)
#define D_MAP_CLEAR(handle)       (handle->d_map_clear)(handle)

#define D_MAP_PAGE(handle, flags, baddr, busaddr, xmp) \
                 (handle->d_map_page)(handle,flags, baddr, busaddr, xmp)

#define D_UNMAP_PAGE(handle, bus_addr) \
   if (handle->d_unmap_page != NULL) (handle->d_unmap_page)(handle, bus_addr)

#define D_MAP_LIST(handle, flags, minxfer, virt_list, bus_list) \
             (handle->d_map_list)(handle, flags, minxfer, virt_list,bus_list)

#define D_UNMAP_LIST(handle, bus_list) \
   if (handle->d_unmap_list != NULL)(handle->d_unmap_list)(handle, bus_list)

#define D_MAP_SLAVE(handle, flags, minxfer, vlist, chan_flags) \
      (handle->d_map_slave)(handle, flags, minxfer, vlist, chan_flags)

#define D_UNMAP_SLAVE(handle) \
      (handle->d_unmap_slave != NULL) ? \
      (handle->d_unmap_slave)(handle) : DMA_SUCC

#define D_MAP_DISABLE(handle)  (handle->d_map_disable)(handle)

#define D_MAP_ENABLE(handle)   (handle->d_map_enable)(handle)

#define D_SYNC_MEM(handle, bus_list) \
      (handle->d_sync_mem != NULL) ? \
```

```
      (handle->d_sync_mem)(handle, bus_list) : DMA_SUCC
#define D_MAP_ATTR(handle, cmd, attr, attr_size) \
      (handle->d_map_attr)(handle, cmd, attr, attr_size)
```

## The d_map Return Code Map

The table in this section describes the possible return codes and requirements for the **d_map** interfaces that map memory for DMA.

*Table 1. Return codes and requirements for the **d_map** interfaces*

| Return Codes | d_map_page | d_map_list | d_map_slave |
|---|---|---|---|
| DMA_SUCC | Page mapped successfully, busaddr contains the mapped bus address. **d_unmap_page** must be called to free any resources associated with the mapping. | List mapped successfully. bus_list describes list of mapped bus addresses.**d_unmap_list** must be called to free any resources associated with the mapping. | List mapped successfully. Slave DMA Controller initialized for the required transfer. **d_unmap_slave** must be called to free any resources associated with the mapping. |
| DMA_NORES | Not enough resources to map the page. No mapping is performed **d_unmap_page** must not be called. | Not enough resources to map the entire list. A partial mapping is possible. **d_unmap_list** must be called to free any resources associated with the mapping. | Not enough resources to map the entire list. A partial mapping is possible. **d_unmap_slave** must be called to free any resources associated with the mapping. |
| DMA_NOACC | No access to the page. No mapping is performed. **d_unmap_page** must not be called. | No access to a page in the list. No mapping is performed. **d_unmap_list** must not be called. | No access to a page in the list. No mapping is performed. **d_unmap_slave** must not be called. |
| DMA_DIOFULL | Does not apply. | bus_list is exhausted. Successful partial mapping as indicated. **d_unmap_list** must be called when finished with the partial mapping. | Does not apply. |
| DMA_BAD_MODE | Does not apply. | Does not apply. | Requested channel modes are not supported. |

## Using the dio Structure

A device driver can use the `dio` structure in many ways.

It can be used to:

- Pass a list of virtual addresses and lengths of buffers to the `d_map_list` and `d_map_slave` services.
- Receive the resulting list of bus addresses (`d_map_list` only) for use by the device in the data transfer.

  **Note:** The driver does not need a `dio` bus list for calls to `d_map_slave` because the address generation for workers is hidden.

Typically, a device driver provides a dio structure that contains only one virtual buffer and one length in the list. If the virtual buffer length spans many pages, the bus address list contains multiple entries that reflect the physical locations of the virtually contiguous buffer. The driver can provide multiple virtual buffers in the virtual list. The driver can then place many buffer requests in one I/O operation.

The device driver is responsible for allocating the storage for all the dio lists it needs. For more information, see the DIO_INIT and DIO_FREE macros in the sys/dma.h header file. The driver must have at least two dio structures. One is needed for passing in the virtual list. Another is needed to accept the resulting bus list. The driver can have many dio lists if it plans to have multiple outstanding I/O commands to its device. The length of each list is dependent on the use of the device and driver. The virtual list needs as many elements as the device could place in one operation at the same time. A formula for estimating how many elements the bus address list needs is the sum of each of the virtual buffers lengths divided by page size plus 2. Or,

```
sum [i=0 to n] ((vlist[i].length / PSIZE) + 2).
```

This formula handles a worst-case situation. For a contiguous virtual buffer that spans multiple pages, each physical page is discontiguous, and neither the starting nor ending addresses are page-aligned.

If the d_map_list service runs out of space when filling in the dio bus list, a DMA_DIOFULL error is returned to the device driver and the *bytes_done* field of the dio virtual list is set to the number of bytes successfully mapped in the bus list. This byte count is guaranteed to be a multiple of the minxfer field provided to the d_map_list or d_map_slave services. Also, the resid_iov field of the virtual list is set to the index of the first d_iovec entry that represents the remainder of iovecs that could not be mapped.

The device driver can:

- Initiate a partial transfer on its device and leave the remainder on its device queue.

  If the driver chooses not to initiate the partial transfer, it must still make a call to d_unmap_list to undo the partial mapping.
- Make another call to the d_map_list with new dio lists for the remainder and setup its device for the full transfer that was originally intended.

If d_map_list or d_map_slave encounter an access violation on a page within the virtual list, then a DMA_NOACC error is returned to the device driver and the bytes_done field of the dio virtual list is set to the number of bytes that preceded the faulting iovec. In this case, the resid_iov field is set to the index of the d_iovec entry that encountered the violation. From this information, the driver can determine which virtual buffer contained the faulting page and fail that request back to the originator.

**Note:** If the DMA_NOACC error is returned, the bytes_done count is not guaranteed to be a multiple of the minxfer field provided to the d_map_list or d_map_slave services, and no partial mapping is done. For workers, setup of the address generation hardware is not done. For controllers, the bus list is undefined. If the driver desires a partial transfer, it must make another call to the mapping service with the dio list adjusted to not include the faulting buffer.

If either the d_map_list or d_map_slave services run out of resources when mapping a transfer, a DMA_NORES error is returned to the device driver. In this case, the bytes_done field of the dio virtual list is set to the number of bytes that were successfully mapped in the bus list. This byte count is guaranteed to be a multiple of the minxfer field provided to the d_map_list or d_map_slave services. Also, the resid_iov field of the virtual list is set to the index of the first d_iovec of the remaining iovecs that could not be mapped. The device driver can:

- Initiate a partial transfer on its device and leave the remainder on its device queue

  If the driver chooses not to initiate the partial transfer, it still must make a call to d_unmap_list or d_unmap_slave (for workers) to undo the partial mapping.
- Choose to leave the entire request on its device queue and wait for resources to free up (for example, after a device interrupt from a previous operation).

**Note:** If the DMA_ENABLE_64 flag is indicated on the d_map_init call or the **d_map_init_ext** call, the programming model is the same with one exception. The dio_64 and d_iovec_64 structures are used in addition to 64-bit address fields on d_map_page and d_unmap_page calls.

## Fields of dio

The only field of the bus list that a device driver modifies is the total_iovecs field to indicate how many elements are available in the list. The device driver never changes any of the other fields in the bus list.

The device driver uses the bus list to set up its device for the transfer. The bus list is provided to the d_unmap_list service to unmap the transfer. The d_map_list service sets the used_iovecs field to indicate how many elements it filled out. The device driver sets up all of the fields in the virtual list except for the bytes_done and resid_iov fields. These fields are set by the mapping service.

## Using DMA_CONTIGUOUS

The DMA_CONTIGUOUS flag in the d_map_init service or the **d_map_init_ext** service is the preferred way for the drivers to ask for contiguous bus addresses.

The other way is the old model of drivers explicitly using rmalloc() to guarantee contiguous allocation during boot. However, with the advent of PCI Hot Plug devices, the rmalloc reservation does not add a device after boot. If a PowerPC® driver determines that the device was dynamically added, the driver can use the DMA_CONTIGUOUS flag to ensure that a contiguous list of bus addresses is generated because no prior resources were reserved with rmalloc.

## Using DMA_NO_ZERO_ADDR

DMA_NO_ZERO_ADDR is supplied on the d_map_init service or the **d_map_init_ext** service to prevent d_map_page and d_map_list from giving out bus address zero to this d_handle.

Because many off-the-shelf PCI devices are not tested for bus address of zero, such devices might not work. Striking out bus address 0 causes a driver's mappable memory to shrink by one I/O page (4 KB). On some systems, using the flag would cause the d_map_init service or the **d_map_init_ext** service to fail even if there is not an error condition. In such a case, the driver should call the d_map_init service or the **d_map_init_ext** service without the flag and then check the bus address to see whether zero falls in its range of addresses. The driver can do this by mapping all of its range and checking for address 0. Such a check should be done at the driver initialization time. If bus address 0 is assigned to the driver, it can leave it mapped for the life of the driver and unmap all other addresses. This guarantees that address 0 is not assigned to it again.

## Using DMA_MAXMIN_MAPSPACE

The DMA_MAXMIN_MAPSPACE flag is supplied on the d_map_init service or the **d_map_init_ext** service to prevent the d_map_init service or the **d_map_init_ext** service from allocating more DMA space for the DMA handle than what the device driver requests.

The device drivers request a specific amount of DMA space by specifying a DMA_MAXMIN_* flag on the **d_map_init** service or by specifying the **d_info_t.di_max_mapspace** flag on the **d_map_init_ext** service. Because some device drivers support non-page-aligned DMA transfers of the specified maximum DMA space, by default the d_map_init service or the **d_map_init_ext** service allocates at least one additional page of DMA space.

Device drivers that use the DMA_MAXMIN_MAPSPACE flag cannot support non-page-aligned DMA transfers of the specified maximum DMA space. The DMA_MAXMIN_MAPSPACE flag indicates that the value of the maximum DMA space represents the amount of mappable address space the device driver requires, rather than the maximum transfer value.

## Using DMA_INIT_STMAP_SUPPORT

To indicate that the driver supports short-term mapping and long-term mapping, you can specify the *flags* parameter of the kernel services that are listed in this section with the corresponding values in the table.

*Table 2. Short-term mapping and long-term mapping support*

| Kernel service | The value of the *flags* parameter |
|---|---|
| **d_map_init_ext** | **DMA_INIT_STMAP_SUPPORT** |
| **d_map_page** | **DMA_STMAP** |
| **d_map_list** | **DMA_STMAP** |

## Sample pseudo-code for the PCI drivers

The example in this section shows a sample pseudo-code for the PCI drivers.

```
dd_initialization:

        determine bus type for device from configuration information
        determine 64 vs. 32-bit capabilities from configuration information

        di_info.di_bid = bid
        di_info.di_flags = DMA_MASTER | flags (except DMA_MAXMIN_* flag);
        di_info.di_bus_flags = 0;
        di_info.di_channel = 0;
        di_info.di_min_mapspace = min_mapspace;
        di_info.di_des_mapspace = DMA_MAXMIN_* value;
        di_info.di_max_mapspace = max_dma_space ;
        call "rc = D_MAP_INIT_EXT(&di_info, sizeof(di_info), handle)"
        if handle == DMA_FAIL OR rc != DMA_SUCC
                could not configure

dd_start_io:

        if single page or less transfer
               call "result = D_MAP_PAGE(handle, baddr,busaddr, xmem)"
                if result == DMA_NORES
                        no resources, leave request on device queue
                else if result == DMA_NOACC
                         no access to page, fail request
                else
                         program device for transfer using busaddr
        else
                create dio list of virtual addresses involved in transfer
                call "result = D_MAP_LIST(handle, flags, minxfer, list, blist)"
                if result == DMA_NORES
                       not enough resource, either initiate partial transfer
                       and leave remainder on queue or leave entire request
                       on the queue and call d_unmap_list to unmap the
                       partial transfer.
               else if result == DMA_NOACC
                       use bytes_done to pinpoint failing buffer and
                       fail corresponding request adjust virtual list and
                      call d_map_list again
                else if result == DMA_DIOFULL
                       ran out of space in blist. either initiate partial
                       transfer and leave remainder on queue or leave entire
                       request on the queue and call d_unmap_list to
                       unmap the partial transfer.
               else
                       program device for scatter/gather transfer using blist

dd_finish_io:

               if single page or less transfer
                       call "D_UNMAP_PAGE(handle, busaddr)"
               else
                       call "D_UNMAP_LIST(handle, blist)"

dd_unconfigure:

               call "D_MAP_CLEAR(handle)"
```

## Sample Pseudo-code for the ISA worker drivers

The example in this section shows a sample pseudo-code for the ISA worker drivers.

```
dd_initialization:

               determine bus type for device from configuration information
               call "handle = D_MAP_INIT(bid, DMA_SLAVE, bus_flags, channel)"
               if handle == DMA_FAIL
                               could not configure
               else
                               call "D_MAP_ENABLE(handle)" (if necessary)

dd_start_io:

               create dio list of virtual addresses involved in transfer
               call "result = D_MAP_SLAVE(handle, flags, minxfer, vlist,
chan_flags)"
               if result == DMA_NORES
                       not enough resource, either initiate partial transfer
                       and leave remainder on queue or leave entire request
                       on the queue and call d_unmap_slave to unmap the
                               partial transfer.
               else if result == DMA_NOACC
                               use bytes_done to pinpoint failing buffer and
                               fail corresponding request
                               adjust virtual list and call d_map_slave again
               else
                               program device to initiate transfer

dd_finish_io:

               call "error = D_UNMAP_SLAVE(handle)"
               if error
                               log
                               retry, or fail

dd_unconfigure:

               call "D_MAP_DISABLE(handle)" (if necessary)
               call "D_MAP_CLEAR(handle)"
```

## Page Protection Checking and Enforcement

Page protection checking is performed by the d_map_page, d_map_list, and d_map_slave services for each page of a requested transfer.

If the intended direction of a transfer is from the device to the memory, the page access permissions must support writing to the page. If the intended direction of a transfer is from the memory to the device, the page access permissions only require reading from the page. In the case of a protection violation, a DMA_NOACC return code is returned from the services in the form of an error code and no mapping for the DMA transfer is performed.

The DMA_BYPASS flag lets a device driver bypass the access checking functionality of these services. This should only be used for global system buffers such as mbufs or other command, control, and status buffers used by a device driver. Also, the DMA buffers *must* be pinned before the DMA transfer begins and can only be unpinned after the DMA transfer is complete.

## Short term mapping

Specifying the **DMA_STMAP** flag on both the **d_map_page** service and the **d_map_list** service is the preferred method for a caller to indicate whether the mapping is for a short term.

Indicating that a mapping is for a short term may allow the driver to map additional memory beyond the amount of I/O mappable memory that is reserved by the **d_map_init_ext** kernel service. The caller must pass the **DMA_INIT_STMAP_SUPPORT** flag on the **d_map_init_ext** service to recognize the **DMA_STMAP** flag.

## A comparison of PCI and ISA devices

The ISA bus has the unique concepts that are listed in this section that do not apply to the PCI bus.

- Enabling and disabling a DMA channel applies only to the ISA bus and devices. Therefore, `d_map_enable` and `d_map_disable` services cannot be used by PCI device drivers.
- Controller and worker devices are not applicable to the PCI bus. On a PCI bus, every device acts as controller.

Only ISA worker devices are supported (ISA controllers are not supported). For such ISA worker devices, the PCI-to-ISA bridge acts as the PCI controller and initiates DMA on behalf of the ISA worker devices. Because the PCI devices are always controllers, `d_map_slave` and `d_unmap_slave` services cannot be used by PCI device drivers. By the same token, the `DMA_SLAVE` flag cannot be supplied on the `d_map_init` service or the **d_map_init_ext** service by a PCI device driver. If `DMA_SLAVE` is used by a PCI driver, the `d_map_init` service or the **d_map_init_ext** service returns `DMA_FAIL`.

## d_align and d_roundup

The **d_align** service (provided in **libsys.a**) returns the alignment value required for starting a buffer on a processor cache line boundary. The **d_roundup** service (also provided in **libsys.a**) can be used to round the desired DMA buffer length up to a value that is an integer number of cache lines.

These two services use buffers for DMA alignment on a cache line boundary. The buffers are also used to allocate DMA in whole multiples of the cache line size so that the buffer is not split across processor cache lines. This reduces the possibility of consistency problems because of DMA and also minimizes the number of cache lines that must be flushed or invalidated when used for DMA. For example, these services can be used to provide alignment as follows:

```
align = d_align();
buffer_length = d_roundup(required_length);
buf_ptr = xmalloc(buffer_length, align, kernel_heap);
```

## Using 64-bit DMA transfer

The 64-bit direct memory access (DMA) transfer enables devices to extend beyond the 2 GB size limit and enables DMA windows to be placed above the 4 GB limit in system memory. The 64-bit DMA transfer is only supported on PCI-E devices that have support for dynamic DMA Windows. A device that uses the 64-bit DMA transfer forfeits the use of the 32-bit DMA transfer on that device.

To use the 64-bit DMA support, the **d_map_query** service must be called along with the **DDW_QUERY** command.

The **d_map_query** service returns the maximum number of contiguous pages available for a window, and the supported page sizes. The driver can find the maximum size allowed by multiplying the number of pages available and their desired supported page size. After a suitable space is found, the driver must call the **d_map_init_ext** service along with the **DMA_PRIME** flag to create a DDW. To create a DMA handle, the device driver must specify the **DMA_HUGE** flag along with the **DMA_PRIME** flag or both flags along with the **d_map_init_ext** service. The calls to the **d_map_init_ext** service, where both **DMA_PRIME** and **DMA_HUGE** flags are specified, uses the same size input for both the DDW and DMA handle.

The device driver can be bypass using the **DMA_PRIME** flag to create a window by specifying the **DMA_HUGE** flag in the **d_map_init_ext** service. If no DDW window was created for the slot previously, then one is created for it. However, the 32-bit DMA handles can not be allocated at the time.

### *Using DMA_HUGE*

The **DMA_HUGE** flag creates a 64-bit direct memory access (DMA) handle. The 64-bit DMA handle is allocated within a dynamic DMA Windows. The **DMA_HUGE** flag handles up to 512 GB depending on the dynamic DMA Window size. Using the **d_map_query** service along with the **DMA_QUERY** flag specifies the maximum amount of DMA space available in the currently existing windows.

If the **DMA_HUGE** flag is specified before the **DMA_PRIME** flag, the **DMA_HUGE** flag then searches through any current dynamic DMA Windows for space available. If the **DMA_PRIME** flag is not specified, it creates one of maximum size and allocates a handle of specified size to the DMA space.

### *Using DMA_PRIME*

The **DMA_PRIME** flag is only supported by the 64-bit direct memory access (DMA) transfer and indicates that a new dynamic DMA Window is required. If the **DMA_PRIME** flag is used along with the **DMA_HUGE** flag, both the new DDW window and the DMA handle is of the same size.

If only the **DMA_PRIME** flag is specified, a dynamic DMA Window is created. The **d_map_init_ext** service along with the specified **DMA_HUGE** flag must then be called to allocate a DMA handle. The **DMA_PRIME** flag creates up to a 512 GB window depending on system resources.

Using the **d_map_query** service along with the **DDW_QUERY** command specifies the maximum amount of pages available for the window. The amount of pages multiplied by the desired page size for the window specifies the amount of space available for the window.

## Using DMA_HUGE

The **DMA_HUGE** flag creates a 64-bit DMA handle and must be used along with or after a call specifying the **DMA_PRIME** flag. The 64-bit DMA handle is allocated within a DDW.

The **DMA_HUGE** flag enables handles up to 512 GB depending on the DDW size. Using the **d_map_query** service along with the **DMA_QUERY** flag specifies the maximum amount of DMA space available in the currently existing windows.

## Using DMA_PRIME

The **DMA_PRIME** flag must be used before or along with the **DMA_HUGE** flag. The **DMA_PRIME** flag is only supported by the 64-bit DMA transfer and indicates that a new DDW is desired.

If the **DMA_PRIME** flag is used along with the **DMA_HUGE** flag, both the new DDW window and the DMA handle are of the same size.

If only the **DMA_PRIME** flag is specified, a DDW is created. The **d_map_init_ext** service along with the **DMA_HUGE** flag specified must then be called to allocate a DMA handle. The **DMA_PRIME** flag supports up to a 512 GB window to be created depending on system resources.

Using the **d_map_query** service along with the **DDW_QUERY** command specifies the maximum amount of pages available for the window. The amount of pages returned by the words multiplied by the desired page size for the window specifies the amount of space available for the window.

# Kernel Extension and Device Driver Management Services

The kernel provides a set of program and device driver management services. These services include kernel extension loading and unloading services and device driver binding services.

Services that allow kernel extensions to be notified of base kernel configuration changes, user-mode exceptions, and process state changes are also provided.

## Kernel Extension Loading and Unloading Services

The program and device driver management services provide the kernel extension loading, unloading, and query services.

The **kmod_load**, **kmod_unload**, and **kmod_entrypt** services provide kernel extension loading, unloading, and query services. User-mode programs and kernel processes can use the **sysconfig** subroutine to invoke the **kmod_load** and **kmod_unload** services. The **kmod_entrypt** service returns a pointer to a kernel extension's entry point.

The **kmod_load**, **kmod_unload** services can be used to dynamically alter the set of routines loaded into the kernel based on system configuration and application demand. Subsystems and device drivers can use these services to load large, seldom-used routines on demand.

## Other Kernel Extension and Device Driver Management Services

The device driver binding services are **devswadd**, **devswdel**, **devswchg**, and **devswqry**.

The **devswadd**, **devswdel**, and **devswchg** services are used to add, remove, or modify device driver entries in the dynamically-managed device switch table. The **devswqry** service is used to obtain information about a particular device switch table entry.

Some kernel extensions might be sensitive to the settings of base kernel runtime configurable parameters that are found in the **var** structure defined in the **/usr/include/sys/var.h** file. These parameters can be set automatically during system boot or at runtime by a privileged user. Kernel extensions can register or unregister a configuration notification routine with the **cfgnadd** and **cfgndel** kernel services. Each time the **sysconfig** subroutine is used to change base kernel tunable parameters found in the **var** structure, each registered configuration notification routine is called.

The **prochadd** and **prochdel** kernel services allow kernel extensions to be notified when any process in the system has a state transition, such as being created, exiting, or being swapped in or swapped out.

The **uexadd** and **uexdel** kernel services give kernel extensions the capability to intercept user-mode exceptions. A user-mode exception handler can use this capability to dynamically reassign access to single-use resources or to clean up after some particular user-mode error. The associated **uexblock** and **uexclear** services can be used by these handlers to block and resume process execution when handling these exceptions.

The **getexcept** kernel service is used by device drivers to obtain detailed information about exceptions that occur during I/O bus access. The **getexcept** service can also be used by any exception handler requiring more information about an exception that has occurred. The **selreg** kernel service is used by file select operations to register unsatisfied asynchronous poll or select event requests with the kernel. The **selnotify** kernel service provides the same functionality as the **selwakeup** service found on other operating systems.

The **iostadd** and **iostdel** services are used by tty and disk device drivers to register device activity reporting structures to be used by the **iostat** and **vmstat** commands.

The **getuerror** and **setuerror** services allow kernel extensions to read or set the `ut_error` field for the current thread. This field can be used to pass an error code from a system call function to an application program, because kernel extensions do not have direct access to the application's **errno** variable.

## List of Kernel Extension and Device Driver Management Kernel Services

This section lists the kernel program and device driver management kernel services with their description.

| Item | Description |
|---|---|
| **cfgnadd** | Registers a notification routine to be called when system-configurable variables are changed. |
| **cfgndel** | Removes a notification routine for receiving broadcasts of changes to system configurable variables. |
| **devdump** | Calls a device driver dump-to-device routine. |
| **devstrat** | Calls a block device driver's strategy routine. |
| **devswadd** | Adds a device entry to the device switch table. |
| **devswchg** | Alters a device switch entry point in the device switch table. |
| **devswdel** | Deletes a device driver entry from the device switch table. |
| **devswqry** | Checks the status of a device switch entry in the device switch table. |
| **getexcept** | Allows kernel exception handlers to retrieve additional exception information. |
| **getuerror** | Allows kernel extensions to read the `ut_error` field for the current thread. |

| Item | Description |
|---|---|
| **iostadd** | Registers an I/O statistics structure used for updating I/O statistics reported by the **iostat** subroutine. |
| **iostdel** | Removes the registration of an I/O statistics structure used for maintaining I/O statistics on a particular device. |
| **kmod_entrypt** | Returns a function pointer to a kernel module's entry point. |
| **kmod_load** | Loads an object file into the kernel or queries for an object file already loaded. |
| **kmod_unload** | Unloads a kernel object file. |
| **prochadd** | Adds a system wide process state-change notification routine. |
| **prochdel** | Deletes a process state change notification routine. |
| **selreg** | Registers an asynchronous poll or select request with the kernel. |
| **selnotify** | Wakes up processes waiting in a **poll** or **select** subroutine or the **fp_poll** kernel service. |
| **setuerror** | Allows kernel extensions to set the `ut_error` field for the current thread. |
| **uexadd** | Adds a system wide exception handler for catching user-mode process exceptions. |
| **uexblock** | Makes the currently active kernel thread not runnable when called from a user-mode exception handler. |
| **uexclear** | Makes a kernel thread blocked by the **uexblock** service runnable again. |
| **uexdel** | Deletes a previously added system-wide user-mode exception handler. |

# Locking Kernel Services

The kernel services can be locked using any of the methods that are described in this overview.

**Related concepts**

Kernel Environment
The kernel is dynamically extendable and can be expanded by adding routines that belong to any of the functional classes.

Kernel Services
*Kernel services* are routines that provide the runtime kernel environment to programs executing in kernel mode. Kernel extensions call kernel services, which resemble library routines. In contrast, application programs call library routines.

System Calls
A system call is a routine that allows a user application to request actions that require special privileges. Adding system calls is one of several ways to extend the functions provided by the kernel.

## Lock Allocation and Other Services

The lock allocation services that are described in this section allocate and free internal operating system memory for simple and complex locks, or check if the caller owns a lock.

| Item | Description |
|---|---|
| **lock_alloc** | Allocates system memory for a simple or complex lock. |
| **lock_free** | Frees the system memory of a simple or complex lock. |
| **lock_mine** | Checks whether a simple or complex lock is owned by the caller. |

## Simple Locks

Simple locks are exclusive-write, non-recursive locks that protect thread-thread or thread-interrupt critical sections. Simple locks are preemptable, meaning that a kernel thread can be preempted by another, higher priority kernel thread while it holds a simple lock.

The simple lock kernel services are:

| Item | Description |
|---|---|
| **simple_lock_init** | Initializes a simple lock. |
| **simple_lock**, **simple_lock_try** | Locks a simple lock. |
| **simple_unlock** | Unlocks a simple lock. |

On a multiprocessor system, simple locks that protect thread-interrupt critical sections must be used in conjunction with interrupt control in order to serialize execution both within the executing processor and between different processors. On a uniprocessor system interrupt control is sufficient; there is no need to use locks. The following kernel services provide appropriate locking calls for the system on which they are executed:

| Item | Description |
|---|---|
| **disable_lock** | Raises the interrupt priority, and locks a simple lock if necessary. |
| **unlock_enable** | Unlocks a simple lock if necessary, and restores the interrupt priority. |

Using the **disable_lock** and **unlock_enable** kernel services to protect thread-interrupt critical sections (instead of calling the underlying interrupt control and locking kernel services directly) ensures that multiprocessor-safe code does not make unnecessary locking calls on uniprocessor systems.

Simple locks are spin locks; a kernel thread that attempts to acquire a simple lock may spin (busy-wait: repeatedly execute instructions which do nothing) if the lock is not free. The table shows the behavior of kernel threads and interrupt handlers that attempt to acquire a busy simple lock.

| Caller | Owner is Running | Owner is Sleeping |
|---|---|---|
| Thread (with interrupts enabled) | Caller spins initially; it sleeps if the maximum spin threshold is crossed. | Caller sleeps immediately. |
| Interrupt handler or thread (with interrupts disabled) | Caller spins until lock is acquired. | Caller spins until lock is freed (must not happen). |

**Note:** On uniprocessor systems, the maximum spin threshold is set to one, meaning that that a kernel thread will never spin waiting for a lock.

A simple lock that protects a thread-interrupt critical section must never be held across a sleep, otherwise the interrupt could spin for the duration of the sleep, as shown in the table. This means that such a routine must not call any external services that might result in a sleep. In general, using any kernel service which is callable from process level may result in a sleep, as can accessing unpinned data. These restrictions do not apply to simple locks that protect thread-thread critical sections.

The lock word of a simple lock must be located in pinned memory if simple locking services are called with interrupts disabled.

## Complex Locks

Complex locks are read-write locks that protect thread-thread critical sections. Complex locks are preemptable, meaning that a kernel thread can be preempted by another, higher priority kernel thread while it holds a complex lock.

The complex lock kernel services are:

| Item | Description |
|---|---|
| **lock_init** | Initializes a complex lock. |
| **lock_islocked** | Tests whether a complex lock is locked. |
| **lock_done** | Unlocks a complex lock. |
| **lock_read**, **lock_try_read** | Locks a complex lock in shared-read mode. |
| **lock_read_to_write**, **lock_try_read_to_write** | Upgrades a complex lock from shared-read mode to exclusive-write mode. |
| **lock_write**, **lock_try_write** | Locks a complex lock in exclusive-write mode. |
| **lock_write_to_read** | Downgrades a complex lock from exclusive-write mode to shared-read mode. |
| **lock_set_recursive** | Prepares a complex lock for recursive use. |
| **lock_clear_recursive** | Prevents a complex lock from being acquired recursively. |

By default, complex locks are not recursive (they cannot be acquired in exclusive-write mode multiple times by a single thread). A complex lock can become recursive through the **lock_set_recursive** kernel service. A recursive complex lock is not freed until **lock_done** is called once for each time that the lock was locked.

Complex locks are not spin locks; a kernel thread that attempts to acquire a complex lock may spin briefly (busy-wait: repeatedly execute instructions which do nothing) if the lock is not free. The table shows the behavior of kernel threads that attempt to acquire a busy complex lock:

| Current Lock Mode | Owner is Running and no Other Thread is Asleep on This Lock | Owner is Sleeping |
|---|---|---|
| Exclusive-write | Caller spins initially, but sleeps if the maximum spin threshold is crossed, or if the owner later sleeps. | Caller sleeps immediately. |
| Shared-read being acquired for exclusive-write | Caller sleeps immediately. | |
| Shared-read being acquired for shared-read | Lock granted immediately | |

**Note:**

1. On uniprocessor systems, the maximum spin threshold is set to one, meaning that a kernel thread will never spin waiting for a lock.
2. The concept of a single owner does not apply to a lock held in shared-read mode.

## Lockl Locks

Lockl locks are exclusive-access and recursive locks.

**Note:** Lockl locks (previously called conventional locks) are only provided to ensure compatibility with existing code. New code should use simple or complex locks.

The lockl lock kernel services are:

| Item | Description |
|---|---|
| **lockl** | Locks a conventional lock. |
| **unlockl** | Unlocks a conventional lock. |

A thread which tries to acquire a busy lockl lock sleeps immediately.

The lock word of a lockl lock must be located in pinned memory if the lockl service is called with interrupts disabled.

### Atomic Operations

Atomic operations are sequences of instructions that guarantee atomic accesses and updates of shared single word variables. This means that atomic operations cannot protect accesses to complex data structures in the way that locks can, but they provide a very efficient way of serializing access to a single word.

The atomic operation kernel services are:

| Item | Description |
|---|---|
| **fetch_and_add** | Increments a single word variable atomically. |
| **fetch_and_and**, **fetch_and_or** | Manipulates bits in a single word variable atomically. |
| **compare_and_swap** | Conditionally updates or returns a single word variable atomically. |

Single word variables accessed by atomic operations must be aligned on a full word boundary, and must be located in pinned memory if atomic operation kernel services are called with interrupts disabled.

## File Descriptor Management Services

The File Descriptor Management services are supplied by the logical file system for creating, using, and maintaining file descriptors. These services allow for the implementation of system calls that use a file descriptor as a parameter, create a file descriptor, or return file descriptors to calling applications.

The following are the File Descriptor Management services:

| Item | Description |
|---|---|
| **ufdcreate** | Allocates and initializes a file descriptor. |
| **ufdhold** | Increments the reference count on a file descriptor. |
| **ufdrele** | Decrements the reference count on a file descriptor. |
| **ufdgetf** | Gets a file structure pointer from a held file descriptor. |
| **getufdflags** | Gets the flags from a file descriptor. |
| **setufdflags** | Sets flags in a file descriptor. |

## Logical File System Kernel Services

The Logical File System services (also known as the **fp_**services) allow processes running in kernel mode to open and manipulate files in the same way that user-mode processes do. Data access limitations make it unreasonable to accomplish these tasks with system calls, so a subset of the file system calls has been provided with an alternate kernel-only interface.

The Logical File System services are one component of the logical file system, which provides the functions required to map system call requests to virtual file system requests. The logical file system is responsible for resolution of file names and file descriptors. It tracks all open files in the system using the file table. The Logical File System services are lower level entry points into the system call support within the logical file system.

Routines in the kernel that must access data stored in files or that must set up paths to devices are the primary users of these services. This occurs most commonly in device drivers, where a lower level device driver must be accessed or where the device requires microcode to be downloaded. Use of the Logical File System services is not, however, restricted to these cases.

A process can use the Logical File System services to establish access to a file or device by calling:

- The **fp_open** service with a path name to the file or device it must access.
- The **fp_opendev** service with the device number of a device it must access.
- The **fp_getf** service with a file descriptor for the file or device. If the process wants to retain access past the duration of the system call, it must then call the **fp_hold** service to acquire a private file pointer.

These three services return a file pointer that is needed to call the other Logical File System services. The other services provide the functions that are provided by the corresponding system calls.

**Related concepts**

Communications I/O Subsystem
The Communication I/O Subsystem consists of one or more physical device handlers (PDHs) that control various communication adapters. The interface to the physical device handlers can support any number of processes, the limit being device-dependent.

SCSI Architectural Model Subsystem
This overview describes the interface between a SCSI Architectural Model (SAM) device driver and a SAM adapter device driver. *SAM* is a set of multiple physical transport types, all of which make use of the SCSI command set.

Small Computer System Interface Subsystem (Parallel SCSI)
This overview describes the interface between a small computer system interface (SCSI) device driver and a SCSI adapter device driver. The information in the article is specific for the parallel SCSI implementation.

Virtual File Systems
The virtual file system (VFS) interface, also known as the v-node interface, provides a bridge between the physical and logical file systems. The information that follows discusses the virtual file system interface, its data structures, and its header files, and explains how to configure a virtual file system.

## Availability of logical file system services

The Logical File System services are available only in the process environment.

See process environment.

In addition, calling the **fp_open** service at certain times can cause a deadlock. The lookup on the file name must acquire file system locks. If the process is already holding any lock on a component of the path, the process will be deadlocked. Therefore, do not use the **fp_open** service when the process is already executing an operation that holds file system locks on the requested path. The operations most likely to cause this condition are those that create files.

## List of Logical File System Kernel Services

This topic lists the logical file system kernel services with their descriptions.

*Table 3. Logical file system kernel services*

| Kernel Service | Description |
|---|---|
| **fp_access** | Checks for access permission to an open file. |
| **fp_close** | Closes a file. |
| **fp_fstat** | Gets the attributes of an open file. |
| **fp_getdevno** | Gets the device number or channel number for a device. |
| **fp_getea** | Reads the value of an extended attribute. |
| **fp_getf** | Retrieves a pointer to a file structure. |

| *Table 3. Logical file system kernel services (continued)* | |
|---|---|
| **Kernel Service** | **Description** |
| **fp_hold** | Increments the open count for a specified file pointer. |
| **fp_ioctl** | Issues a control command to an open device or file. |
| **fp_listea** | Lists the extended attributes associated with a file. |
| **fp_llseek** | Changes the current offset in an open file. This service is used to access offsets greater than 2 GB. |
| **fp_lseek** | Changes the current offset in an open file. |
| **fp_open** | Opens special and regular files or directories. |
| **fp_opendev** | Opens a device special file. |
| **fp_poll** | Checks the I/O status of multiple file pointers, file descriptors, and message queues. |
| **fp_read** | Performs a read on an open file with arguments passed. |
| **fp_readv** | Performs a read operation on an open file with arguments passed in **iovec** elements. |
| **fp_removeea** | Removes an extended attribute. |
| **fp_rwuio** | Performs read or write on an open file with arguments passed in a **uio** structure. |
| **fp_select** | Provides for cascaded, or redirected, support of the select or poll request. |
| **fp_setea** | Sets an extended attribute value. |
| **fp_statea** | Provides information about an extended attribute. |
| **fp_fsync** | Writes changes for a specified range of a file to permanent storage. |
| **fp_write** | Performs a write operation on an open file with arguments passed. |
| **fp_writev** | Performs a write operation on an open file with arguments passed in **iovec** elements. |

## Programmed I/O (PIO) Kernel Services

This overview provides a list of PIO kernel services with a brief description.

| Item | Description |
|---|---|
| **io_map** | Attaches to an I/O mapping |
| **io_map_clear** | Removes an I/O mapping segment |
| **io_map_init** | Creates and initializes an I/O mapping segment |
| **io_unmap** | Detaches from an I/O mapping |

These kernel services are defined in the **adspace.h** and **ioacc.h** header files.

For a list of PIO macros, see Programmed I/O Services in *Understanding the Diagnostic Subsystem for AIX*.

# Memory Kernel Services

The Memory kernel services provide kernel extensions certain abilities that are listed in this section.

- Dynamically allocate and free memory
- Pin and unpin code and data
- Access user memory and transfer data between user and kernel memory
- Create, reference, and change virtual memory objects

## Memory Management Kernel Services

This section lists the memory management services with a brief description.

| Item | Description |
|---|---|
| **init_heap** | Initializes a new heap to be used with kernel memory management services. |
| **xmalloc** | Allocates memory. |
| **xmfree** | Frees allocated memory. |

## ldata Kernel Services

**ldata** stands for "local data". The **ldata** facility supports data localization by allowing kernel subsystems and extensions to create and use **ldata** pools.

The element size plus the initial and maximum number of elements to be contained in the pool is specified while creating the pool. The number of elements in the pool can be dynamically increased up to the maximum. Within the sub-pool the elements are contained such that they are cache-aligned and multiples of cache-line size to promote cache friendliness. Also, the elements in each sub-pool are backed by physical memory local to its corresponding SRAD. Allocation of a storage element is satisfied from the per-SRAD sub-pool on which the caller is located and where the storage element is to be predominately accessed. Deallocation of a storage element returns the element to its associated per-SRAD sub-pool.

Services are provided to allow kernel subsystems and extensions to create, destroy, grow the **ldata** pools. There are a couple of advantages of using **ldata** kernel services over raw xmallocs:

1. Since the memory allocated by **ldata** kernel services are backed by local node memory, it is faster to read and write the **ldata** region on that node.
2. **ldata** elements can be allocated from the interrupt environment. **xmalloc** kernel service cannot be called from the interrupt environment. Of course, there is an upper limit on a given **ldata** pool -- the maximum number of elements asked at **ldata** creation time.

The **ldata** services are:

| Item | Description |
|---|---|
| **ldata_create** | Creates a SRAD-aware pinned storage element pool (**ldata** pool) and returns its handle. |
| **ldata_destroy** | Destroys a **ldata** pool created by **ldata_create**. |
| **ldata_grow** | Expands the count of available pinned storage elements contained within a **ldata** pool. |
| **ldata_alloc** | Allocates a pinned storage element from a **ldata** pool. |
| **ldata_free** | Frees a pinned storage element to a **ldata** pool. |

## Memory Pinning Kernel Services

This overview lists the memory pinning services with a brief description.

| Item | Description |
|---|---|
| **ltpin** | Pins the address range in the system (kernel) space and frees the page space for the associated pages. |
| **ltunpin** | Unpins the address range in system (kernel) address space and reallocates paging space for the specified region. |
| **pin** | Pins the address range in the system (kernel) space. |
| **pincode** | Pins the code and data associated with a loaded object module. |
| **unpin** | Unpins the address range in system (kernel) address space. |
| **unpincode** | Unpins the code and data associated with a loaded object module. |
| **xmempin** | Pins the specified address range in user or system memory, given a valid cross-memory descriptor. |
| **xmemunpin** | Unpins the specified address range in user or system memory, given a valid cross-memory descriptor. |

## User-Memory-Access Kernel Services

In a system call or kernel extension running under a user process, data in the user process can be moved in or out of the kernel using the kernel services listed in this section.

The kernel services that are used to move data in or out of the kernel are the **copyin** and **copyout** services. The **uiomove** service is used for scatter and gather operations. If user data is to be referenced asynchronously, such as from an interrupt handler or a kernel process, the cross memory services must be used.

The User-Memory-Access kernel services are:

| Item | Description |
|---|---|
| **copyin**, **copyin64** | Copies data between user and kernel memory. |
| **copyinstr**, **copyinstr64** | Copies a character string (including the terminating null character) from user to kernel space. |
| **copyout**, **copyout64** | Copies data between user and kernel memory. |
| **fubyte**, **fubyte64** | Fetches, or retrieves, a byte of data from user memory. |
| **fuword**, **fuword64** | Fetches, or retrieves, a word of data from user memory. |
| **subyte**, **subyte64** | Stores a byte of data in user memory. |
| **suword**, **suword64** | Stores a word of data in user memory. |
| **uiomove** | Moves a block of data between kernel space and a space defined by a **uio** structure. |
| **ureadc** | Writes a character to a buffer described by a **uio** structure. |
| **uwritec** | Retrieves a character from a buffer described by a **uio** structure. |

**Note:** The **copyin64**, **copyout64**, **copyinstr64**, **fubyte64**, **fuword64**, **subyte64**, and **suword64** kernel services are defined as macros when compiling kernel extensions on the 64–bit kernel. The macros invoke the corresponding kernel services without the "64" suffix.

## Virtual Memory Management Kernel Services

This section lists the various virtual memory management services along with their descriptions.

These services are described in more detail in “Understanding Virtual Memory Manager Interfaces” on page 72. The Virtual Memory Management services are:

| Item | Description |
|---|---|
| **as_att64** | Selects, allocates, and maps a specified region in the current user address space. |
| **as_det64** | Unmaps and deallocates a region in the specified address space that was mapped with the **as_att64** kernel service. |
| **as_geth64** | Obtains a handle to the virtual memory object for the specified address given in the specified address space. The virtual memory object is protected. |
| **as_getsrval64** | Obtains a handle to the virtual memory object for the specified address given in the specified address space. |
| **as_puth64** | Indicates that no more references will be made to a virtual memory object that was obtained using the **as_geth64** kernel service. |
| **as_seth64** | Maps a specified region in the specified address space for the specified virtual memory object. |
| **vm_att** | Maps a specified virtual memory object to a region in the current address space. |
| **vm_cflush** | Flushes the processor's cache for a specified address range. |
| **vm_det** | Unmaps and deallocates the region in the current address space that contains a given address. |
| **vm_flushp** | Flushes the specified range of pages. |
| **vm_galloc** | Allocates a region of global memory in the 64-bit kernel. |
| **vm_gfree** | Frees a region of global memory in the kernel previously allocated with the **vm_galloc** kernel service. |
| **vm_handle** | Constructs a virtual memory handle for mapping a virtual memory object with specified access level. |
| **vm_invalidatep** | Releases page frames in the specified range for a non-journaled persistent segment or client segment. |
| **vm_ioaccessp** | Initiates asynchronous page-in or page-out for the specified range of pages . |
| **vm_makep** | Makes a page in client storage. |
| **vm_mount** | Adds a file system to the paging device table. |
| **vm_mounte** | Adds a file system with a thread-level strategy routine to the paging device table. |
| **vm_move** | Moves data between a virtual memory object and a buffer specified in the **uio** structure. |
| **vm_mvc** | Reads or writes partial pages of files. |
| **vm_protectp** | Sets the page protection key for a page range. |
| **vm_qmodify** | Determines whether a mapped file has been changed. |
| **vm_qpages** | Returns the number of in-memory page frames associated with the virtual memory object. |

| Item | Description |
|---|---|
| **vm_readp** | Initiates asynchronous page-in for the range of pages specified. |
| **vm_release** | Releases virtual memory resources for the specified address range. |
| **vm_releasep** | Releases virtual memory resources for the specified page range. |
| **vm_segmap** | Creates the segments associated with a range of bytes in a file and attaches them to the kernel's address space. |
| **vm_setdevid** | Modifies the paging-device table entry for a virtual memory object. |
| **vm_thrpgio_pop** | Retrieves the latest per-thread context information in a client file system with a thread page-I/O strategy routine. |
| **vm_thrpgio_push** | Stores the current per-thread context information in a client file system with a thread page-I/O strategy routine. |
| **vm_uiomove** | Moves data between a virtual memory object and a buffer specified in the **uio** structure. |
| **vm_umount** | Removes a file system from the paging device table. |
| **vm_vmid** | Converts a virtual memory handle to a virtual memory object (id). |
| **vm_write** | Initiates page-out for a page range in the address space. |
| **vm_writep** | Initiates page-out for a page range in a virtual memory object. |
| **vms_create** | Creates a virtual memory object of the type and size and limits specified. |
| **vms_delete** | Deletes a virtual memory object. |
| **vms_iowait** | Waits for the completion of all page-out operations for pages in the virtual memory object. |

## Cross-Memory Kernel Services

The cross-memory kernel services allow data to be moved between the kernel and an address space other than the current process address space.

A data area within one region of an address space is attached by calling the **xmattach** service. As a result, the virtual memory object cannot be deleted while data is being moved in or out of pages belonging to it. A cross-memory descriptor is filled out by the **xmattach** service. The attach operation must be done while under a process. When the data movement is completed, the **xmdetach** service can be called. The detach operation can be done from an interrupt handler.

The **xmemin** service can be used to transfer data from an address space to kernel space. The **xmemout** service can be used to transfer data from kernel space to an address space. These routines may be called from interrupt handler level routines if the referenced buffers are in memory.

Cross-memory services provide the **xmemdma64** service to prepare a page for DMA processing. The **xmemdma64** service can be called from the process or interrupt environments. The **xmemdma64** service returns the real address of the page for use in preparing DMA address lists. When the DMA transfer is completed, the **xmemdma64** service must be called again to unhide the page.

Data movement by DMA or an interrupt handler requires that the pages remain in memory. This is ensured by pinning the data areas using the **xmempin** service. This can only be done under a process, because the memory pinning services page-fault on pages not present in memory.

The **xmemunpin** service unpins pinned pages. This can be done by an interrupt handler if the data area is the global kernel address space. It must be done under the process if the data area is in user process space.

The Cross-Memory services are:

| Item | Description |
|---|---|
| **xmattach** | Attaches to a user buffer for cross-memory operations. |
| **xmdetach** | Detaches from a user buffer used for cross-memory operations. |
| **xmemin** | Performs a cross-memory move by copying data from the specified address space to kernel global memory. |
| **xmemout** | Performs a cross-memory move by copying data from kernel global memory to a specified address space. |
| **xmemdma64** | Prepares a page for DMA I/O or processes a page after DMA I/O is complete. Returns 64-bit real address. |
| **xmemzero** | Zeroes a buffer described by a cross-memory descriptor. |

# Understanding Virtual Memory Manager Interfaces

The virtual memory manager supports functions that allow a wide range of kernel extension data operations.

## Virtual Memory Objects

A *virtual memory object* is an abstraction for the contiguous data that can be mapped into a region of an address space. As a data object, it is independent of any address space. The data it represents can be in memory or on an external storage device. The data represented by the virtual memory object can be shared by mapping the virtual memory object into each address space sharing the access, with the access capability of each mapping represented in that address space map.

File systems use virtual memory objects so that the files can be referenced using a mapped file access method. The mapped file access method represents the data through a virtual memory object, and allows the virtual memory manager to handle page faults on the mapped file. When a page fault occurs, the virtual memory manager calls the services supplied by the service provider (such as a virtual file system) to get and put pages. A data provider (such as a file system) maintains any data structures necessary to map between the virtual memory object offset and external storage addressing.

The data provider creates a virtual memory object when it has a request for access to the data. It deletes the virtual memory object when it has no more clients referencing the data in the virtual memory object.

The **vms_create** service is called to create virtual memory objects. The **vms_delete** service is called to delete virtual memory objects.

## Addressing Data

This section describes how data can be addressed in a virtual memory object.

Data in a virtual memory object is made addressable in user or kernel processes through the **shmat** subroutine. A kernel extension uses the **vm_att** kernel service to select and allocate a region in the current (per-process kernel) address space.

The per-process kernel address space initially sees only global kernel memory and the per-process kernel data. The **vm_att** service allows kernel extensions to allocate additional regions. However, this augmented per-process kernel address space does not persist across system calls. The additional regions must be re-allocated with each entry into the kernel protection domain.

The **vm_att** service takes as an argument a virtual memory handle representing the virtual memory object and the access capability to be used. The **vm_handle** service constructs the virtual memory handles.

When the kernel extension has finished processing the data mapped into the current address space, it should call the **vm_det** service to deallocate the region and remove access.

## Moving Data to or from a Virtual Memory Object

Moving data to or from a virtual memory object is an operation on the virtual memory object, not an address space range.

A data provider (such as a file system) can call the **vm_makep** service to cause a memory page to be instantiated. This permits a page of data to be moved into a virtual memory object page without causing the virtual memory manager to page in the previous data contents from an external source.

The **vm_move** and **vm_uiomove** kernel services move data between a virtual memory object and a buffer specified in a **uio** structure. This allows data providers (such as a file system) to move data to or from a specified buffer to a designated offset in a virtual memory object. This service is similar to **uiomove** service, but the trusted buffer is replaced by the virtual memory object, which need not be currently addressable.

## Data Flushing

A kernel extension uses the **vm_write** or **vm_writep** kernel services depending on the addressability to the data area.

A kernel extension can initiate the writing of a data area to external storage with the **vm_write** kernel service, if it has addressability to the data area. The **vm_writep** kernel service can be used if the virtual memory object is not currently addressable.

If the kernel extension needs to ensure that the data is moved successfully, it can wait on the I/O completion by calling the **vms_iowait** service, giving the virtual memory object as an argument.

## Discarding Data

The virtual memory manager interface discards data using the **vm_release** and **vm_releasep** services.

The pages specified by a data range can be released from the underlying virtual memory object by calling the **vm_release** service. The virtual memory manager deallocates any associated paging space slots. A subsequent reference to data in the range results in a page fault.

A virtual memory data provider can release a specified range of pages in a virtual memory object by calling the **vm_releasep** service. The virtual memory object need not be addressable for this call.

## Protecting Data

The service that is listed in this section can change the storage protect keys in a page range.

The **vm_protectp** service can change the storage protect keys in a page range in one client storage virtual memory object. This only acts on the resident pages. The pages are referred to through the virtual memory object. They do not need to be addressable in the current address space. A client file system data provider uses this protection to detect stores of in-memory data, so that mapped files can be extended by storing into them beyond their current end of file.

## Executable Data

If the data moved is to become executable, any data remaining in processor cache must be guaranteed to be moved from cache to memory.

This is because the retrieval of the instruction does not need to use the data cache. The **vm_cflush** service performs this operation.

## Installing Pager Backends

The kernel-extension data providers must provide appropriate routines to be called by the virtual memory manager. These routines move a page-sized block of data into or out of a specified page. These services are also referred to as *pager backends*.

For a local device, the device strategy routine is required. A call to the **vm_mount** service is used to identify the device (through a **dev_t** value) to the virtual memory manager.

For a remote data provider, the routine required is a strategy routine, which is specified in the **vm_mount** service. These strategy routines must run as interrupt-level routines. They must not page fault, and they cannot sleep waiting for locks.

When access to a remote data provider or a local device is removed, the **vm_umount** service must be called to remove the device entry from the virtual memory manager's paging device table.

## Thread Page-I/O Strategy Routine

Pager backends have their strategy routines run at the interrupt level of the Virtual Memory Manager (VMM). Some file systems, however, require that their strategy routines run in the context of the thread that directly causes the page I/O to occur. The VMM accommodates the difference with a blocking, or thread page-I/O strategy routine. If no thread directly causes page I/O to occur, for example, in the case of page replacement, then the strategy routine is invoked in the context of a worker thread from a pool of worker threads that are managed by the VMM.

A file system with a thread page-I/O strategy routine can use all AIX® facilities as part of its handling of page I/O or page protection faults. If the file system intends to re-enter the VMM through client segment page faults or to use VMM services that involve client file segments, the file system must first save the per-thread VMM context by calling the **vm_thrpgio_push** service. The per-thread context can be restored using the **vm_thrpgio_pop** service.

To use a client file system with a thread-level strategy routine, mount the file system with the **vm_mounte** service with its strategy routine specified and the **D_THRPGIO** and **D_ENHANCEDIO** flags set. To remove the file system from the paging device table, call the **vm_umount** service.

## Referenced Routines

The virtual memory manager exports these routines exported to kernel extensions.

| **Services That Manipulate Virtual Memory Objects** | |
|---|---|
| **vm_att** | Selects and allocates a region in the current address space for the specified virtual memory object. |
| **vms_create** | Creates virtual memory object of the specified type and size limits. |
| **vms_delete** | Deletes a virtual memory object. |
| **vm_det** | Unmaps and deallocates the region at a specified address in the current address space. |
| **vm_flushp** | Flushes the specified range of pages. |
| **vm_handle** | Constructs a virtual memory handle for mapping a virtual memory object with a specified access level. |
| **vm_invalidatep** | Releases page frames in the specified range for a non-journaled persistent segment or client segment. |
| **vm_ioaccessp** | Initiates asynchronous page-in or page-out for the specified range of pages . |
| **vms_iowait, vms_iowaitf** | Waits for the completion of all page-out operations in the virtual memory object. |
| **vm_makep** | Makes a page in client storage. |
| **vm_move** | Moves data between the virtual memory object and buffer specified in the **uio** structure. |
| **vm_mvc** | Reads or writes partial pages of files. |

**Services That Manipulate Virtual Memory Objects**

| | |
|---|---|
| **vm_protectp** | Sets the page protection key for a page range. |
| **vm_qpages** | Returns the number of in-memory page frames associated with the virtual memory object. |
| **vm_readp** | Initiates asynchronous page-in for the range of pages specified. |
| **vm_releasep** | Releases page frames and paging space slots for pages in the specified range. |
| **vm_segmap** | Creates the segments associated with a range of bytes in a file and attaches them to the kernel's address space. |
| **vm_setdevid** | Modifies the paging-device table entry for a virtual memory object. |
| **vm_uiomove** | Moves data between the virtual memory object and buffer specified in the **uio** structure. |
| **vm_vmid** | Converts a virtual memory handle to a virtual memory object (id). |
| **vm_writep** | Initiates page-out for a page range in a virtual memory object. |

The following services support address space operations:

| Item | Description |
|---|---|
| **vm_cflush** | Flushes cache lines for a specified address range. |
| **vm_release** | Releases page frames and paging space slots for the specified address range. |
| **vm_write** | Initiates page-out for an address range. |

The following Memory-Pinning kernel services also support address space operations. They are the **pin** and **unpin** services.

**Services That Support Cross-Memory Operations**

Cross Memory Services are listed in "Memory Kernel Services".

**Services that Support the Installation of Pager Backends**

| | |
|---|---|
| **vm_mount** | Allocates an entry in the paging device table. |
| **vm_umount** | Removes a file system from the paging device table. |

## Services that Support 64-bit Processes

The following section lists the services that support 64-bit processes.

| Item | Description |
|---|---|
| **as_att64** | Allocates and maps a specified region in the current user address space. |
| **as_det64** | Unmaps and deallocates a region in the current user address space that was mapped with the as_att64 kernel service. |
| **as_geth64** | Obtains a handle to the virtual memory object for the specified address. |

| Item | Description |
|---|---|
| **as_puth64** | Indicates that no more references will be made to a virtual memory object using the as_geth64 kernel service. |
| **as_seth64** | Maps a specified region for the specified virtual memory object. |
| **as_getsrval64** | Obtains a handle to the virtual memory object for the specified address. |
| **IS64U** | Determines if the current user address space is 64-bit or not. |

## Message Queue Kernel Services

The Message Queue kernel services provide the message queue functions to a kernel extension.

The message queue functions are the same as the **msgctl**, **msgget**, **msgsnd**, and **msgxrcv** subroutines make available to a program executing in user mode. Parameters have been added for moving returned information to an explicit parameter to free the return codes for error code usage. Instead of the error information available in the **errno** global variable (as in user mode), the Message Queue services use the service's return code. The error values are the same, except that a memory fault error (**EFAULT**) cannot occur because message buffer pointers in the kernel address space are assumed to be valid.

The Message Queue services can be called only from the process environment because they prevent the caller from specifying kernel buffers. These services can be used as an Interprocess Communication mechanism to other kernel processes or user-mode processes. See Kernel Extension and Device Driver Management Services for more information on the functions that these services provide.

There are four Message Queue services available from the kernel:

| Item | Description |
|---|---|
| **kmsgctl** | Provides message-queue control operations. |
| **kmsgget** | Obtains a message-queue identifier. |
| **kmsgrcv** | Reads a message from a message queue. |
| **kmsgsnd** | Sends a message using a previously defined message queue. |

## Network Kernel Services

The Network kernel services provide kernel extensions certain abilities that are listed in this overview.

### Address Family Domain and Network Interface Device Driver Kernel Services

The Address Family Domain and Network Interface Device Driver services enable address family domains (Protocols) and network interface drivers to add and remove themselves from network switch tables.

The **if_attach** service and **if_detach** services add and remove network interfaces from the Network Interface List. Protocols search this list to determine an appropriate interface on which to transmit a packet.

Protocols use the **add_input_type** and **del_input_type** services to notify network interface drivers that the protocol is available to handle packets of a certain type. The Network Interface Driver uses the **find_input_type** service to distribute packets to a protocol.

The **add_netisr** and **del_netisr** services add and delete network software interrupt handlers. Address families add and delete themselves from the Address Family Domain switch table by using the **add_domain_af** and **del_domain_af** services. The Address Family Domain switch table is a list of all available protocols that can be used in the **socket** subroutine.

The Address Family Domain and Network Interface Device Driver services are:

| Item | Description |
|---|---|
| **add_domain_af** | Adds an address family to the Address Family domain switch table. |

| Item | Description |
|---|---|
| **add_input_type** | Adds a new input type to the Network Input table. |
| **add_netisr** | Adds a network software interrupt service to the Network Interrupt table. |
| **del_domain_af** | Deletes an address family from the Address Family domain switch table. |
| **del_input_type** | Deletes an input type from the Network Input table. |
| **del_netisr** | Deletes a network software interrupt service routine from the Network Interrupt table. |
| **find_input_type** | Finds the given packet type in the Network Input Interface switch table and distributes the input packet according to the table entry for that type. |
| **if_attach** | Adds a network interface to the network interface list. |
| **if_detach** | Deletes a network interface from the network interface list. |
| **ifunit** | Returns a pointer to the **ifnet** structure of the requested interface. |
| **schednetisr** | Schedules or invokes a network software interrupt service routine. |

## Routing and Interface Address Kernel Services

The Routing and Interface Address services provide protocols with a means of establishing, accessing, and removing routes to remote hosts or gateways. Routes bind destinations to a particular network interface.

The interface address services accept a destination address or network and return an associated interface address. Protocols use these services to determine if an address is on a directly connected network.

The Routing and Interface Address services are:

| Item | Description |
|---|---|
| **ifa_ifwithaddr** | Locates an interface based on a complete address. |
| **ifa_ifwithdstaddr** | Locates the point-to-point interface with a given destination address. |
| **ifa_ifwithnet** | Locates an interface on a specific network. |
| **if_down** | Marks an interface as down. |
| **if_nostat** | Zeroes statistical elements of the interface array in preparation for an attach operation. |
| **rtalloc** | Allocates a route. |
| **rtfree** | Frees the routing table entry |
| **rtinit** | Sets up a routing table entry, typically for a network interface. |
| **rtredirect** | Forces a routing table entry with the specified destination to go through the given gateway. |
| **rtrequest** | Carries out a request to change the routing table. |

## Loopback Kernel Services

The Loopback services enable networking code to be exercised without actually transmitting packets on a network. This is a useful tool for developing new protocols without introducing network variables. Loopback services can also be used to send packets to local addresses without using hardware loopback.

The Loopback services are:

| Item | Description |
|---|---|
| **loifp** | Returns the address of the software loopback interface structure. |

| Item | Description |
|---|---|
| **looutput** | Sends data through a software loopback interface. |

## Protocol Kernel Services

Protocol kernel services provide a means of finding a particular address family as well as a raw protocol handler. The raw protocol handler basically passes raw packets up through sockets so that a protocol can be implemented in user space.

The Protocol kernel services are:

| Item | Description |
|---|---|
| **pfctlinput** | Starts the **ctlinput** function for each configured protocol. |
| **pffindproto** | Returns the address of a protocol switch table entry. |
| **raw_input** | Builds a **raw_header** structure for a packet and sends both to the raw protocol handler. |
| **raw_usrreq** | Implements user requests for raw protocols. |

## Communications Device Handler Interface Kernel Services

This section describes the various Communications Device Handler Interface services.

The Communications Device Handler Interface services provide a standard interface between network interface drivers and communications device handlers. The **net_attach** and **net_detach** services open and close the device handler. Once the device handler has been opened, the **net_xmit** service can be used to transmit packets. Asynchronous `start done` notifications are recorded by the **net_start_done** service. The **net_error** service handles error conditions.

The Communications Device Handler Interface services are:

| Item | Description |
|---|---|
| **add_netopt** | This macro adds a network option structure to the list of network options. |
| **del_netopt** | This macro deletes a network option structure from the list of network options. |
| **net_attach** | Opens a communications I/O device handler. |
| **net_detach** | Closes a communications I/O device handler. |
| **net_error** | Handles errors for communication network interface drivers. |
| **net_sleep** | Sleeps on the specified wait channel. |
| **net_start** | Starts network IDs on a communications I/O device handler. |
| **net_start_done** | Starts the done notification handler for communications I/O device handlers. |
| **net_wakeup** | Wakes up all sleepers waiting on the specified wait channel. |
| **net_xmit** | Transmits data using a communications I/O device handler. |
| **net_xmit_trace** | Traces transmit packets. This kernel service was added for those network interfaces that do not use the **net_xmit** kernel service to trace transmit packets. |

# Process and Exception Management Kernel Services

The process and exception management kernel services provided by the base kernel provide the capability to perform the actions that are listed in this section.

- Create kernel processes

- Register exception handlers
- Provide process serialization
- Generate and handle signals
- Support event waiting and notification

## Creating Kernel Processes

Kernel processes are created by kernel extensions by using the kernel services.

Kernel extensions use the **creatp** and **initp** kernel services to create and initialize a kernel process. The **setpinit** kernel service allow a kernel process to change its parent process from the one that created it to the **init** process, so that the creating process does not receive the death-of-child process signal upon kernel process termination. "Using Kernel Processes" on page 8 provides additional information concerning use of these services.

## Creating Kernel Threads

Kernel threads require fewer system resources than processes, and can start more quickly. Kernel threads are created by kernel extensions.

Kernel extensions use the **thread_create** and **kthread_start** services to create and initialize kernel-only threads. For more information about threads, see "Understanding Kernel Threads" on page 6.

The **thread_setsched** service is used to control the scheduling parameters, priority and scheduling policy, of a thread.

## Kernel Structures Encapsulation

The **getpid** kernel service is used by a kernel extension in either the process or interrupt environment to obtain the process ID of the current process if in the process environment.

The **getpid** kernel service is also used to determine the current execution environment. The **rusage_incr** service provides an access to the **rusage** structure.

The thread-specific **uthread** structure is also encapsulated. The **getuerror** and **setuerror** kernel services should be used to access the `ut_error` field. The **thread_self** kernel service should be used to get the current thread's ID.

## Registering Exception Handlers

The **setjmpx**, **clrjmpx**, and **longjmpx** kernel services allow a kernel extension to register an exception handler by performing the actions that are listed in this section.

- Saving the exception handler's context with the **setjmpx** kernel service
- Removing its saved context with the **clrjmpx** kernel service if no exception occurred
- Starting the next registered exception handler with the **longjmpx** kernel service if it was unable to handle the exception

For more information concerning use of these services, see "Handling Exceptions While in a System Call" on page 29.

## Signal Management

Signals can be posted either to a kernel process or to a kernel thread.

The **pidsig** service posts a signal to a specified kernel process; the **kthread_kill** service posts a signal to a specified kernel thread. A thread uses the **sig_chk** service to poll for signals delivered to the kernel process or thread in the kernel mode.

For more information about signal management, see "Kernel Process Signal and Exception Handling" on page 10.

## Events Management

The event notification services provide support for two types of interprocess communications - primitive and shared.

| Item | Description |
|---|---|
| **Primitive** | Allows only one process thread waiting on the event. |
| **Shared** | Allows multiple processes threads waiting on the event. |

The **et_wait** and **et_post** kernel services support single waiter event notification by using mutually agreed upon event control bits for the kernel thread being posted. There are a limited number of control bits available for use by kernel extensions. If the **kernel_lock** is owned by the caller of the **et_wait** service, it is released and acquired again upon wakeup.

The following kernel services support a shared event notification mechanism that allows for multiple threads to be waiting on the shared event.

| Item | Description |
|---|---|
| **e_assert_wait** | **e_wakeup** |
| **e_block_thread** | e_wakeup_one |
| **e_clear_wait** | e_wakeup_w_result |
| **e_sleep_thread** | e_wakeup_w_sig |

These services support an unlimited number of shared events (by using caller-supplied event words). The following list indicates methods to wait for an event to occur:

- Calling **e_assert_wait** and **e_block_thread** successively; the first call puts the thread on the event queue, the second blocks the thread. Between the two calls, the thread can do any job, like releasing several locks. If only one lock, or no lock at all, needs to be released, one of the two other methods should be preferred.
- Calling **e_sleep_thread**; this service releases a simple or a complex lock, and blocks the thread. The lock can be automatically reacquired at wakeup.

The **e_clear_wait** service can be used by a thread or an interrupt handler to wake up a specified thread, or by a thread that called **e_assert_wait** to remove itself from the event queue without blocking when calling **e_block_thread**. The other wakeup services are event-based. The **e_wakeup** and **e_wakeup_w_result** services wake up every thread sleeping on an event queue; whereas the **e_wakeup_one** service wakes up only the most favored thread. The **e_wakeup_w_sig** service posts a signal to every thread sleeping on an event queue, waking up all the threads whose sleep is interruptible.

The **e_sleep** and **e_sleepl** kernel services are provided for code that was written for previous releases of the operating system. Threads that have called one of these services are woken up by the **e_wakeup**, **e_wakeup_one**, **e_wakeup_w_result**, **e_wakeup_w_sig**, or **e_clear_wait** kernel services. If the caller of the **e_sleep** service owns the **kernel lock**, it is released before waiting and is acquired again upon wakeup. The **e_sleepl** service provides the same function as the **e_sleep** service except that a caller-specified lock is released and acquired again instead of the **kernel_lock**.

## List of Process, Thread, and Exception Management Kernel Services

The Process, Thread, and Exception Management kernel services are listed below.

| Item | Description |
|---|---|
| **clrjmpx** | Removes a saved context by popping the most recently saved jump buffer from the list of saved contexts. |
| **creatp** | Creates a new kernel process. |

| Item | Description |
| --- | --- |
| **e_assert_wait** | Asserts that the calling kernel thread is going to sleep. |
| **e_block_thread** | Blocks the calling kernel thread. |
| **e_clear_wait** | Clears the wait condition for a kernel thread. |
| **e_sleep**, **e_sleep_thread**, or **e_sleepl** | Forces the calling kernel thread to wait for the occurrence of a shared event. |
| **e_sleep_thread** | Forces the calling kernel thread to wait the occurrence of a shared event. |
| **e_wakeup**, **e_wakeup_one**, or **e_wakeup_w_result** | Notifies kernel threads waiting on a shared event of the event's occurrence. |
| **e_wakeup_w_sig** | Posts a signal to sleeping kernel threads. |
| **et_post** | Notifies a kernel thread of the occurrence of one or more events. |
| **et_wait** | Forces the calling kernel thread to wait for the occurrence of an event. |
| **getpid** | Gets the process ID of the current process. |
| **getppidx** | Gets the parent process ID of the specified process. |
| **initp** | Changes the state of a kernel process from idle to ready. |
| **kthread_kill** | Posts a signal to a specified kernel-only thread. |
| **kthread_start** | Starts a previously created kernel-only thread. |
| **limit_sigs** | Changes the signal mask for the calling kernel thread. |
| **longjmpx** | Allows exception handling by causing execution to resume at the most recently saved context. |
| **NLuprintf** | Submits a request to print an internationalized message to the controlling terminal of a process. |
| **pgsignal** | Sends a signal to all of the processes in a process group. |
| **pidsig** | Sends a signal to a process. |
| **rusage_incr** | Increments a field of the **rusage** structure. |
| **setjmpx** | Allows saving the current execution state or context. |
| **setpinit** | Sets the parent of the current kernel process to the **init** process. |
| **sig_chk** | Provides the calling kernel thread with the ability to poll for receipt of signals. |
| **sigsetmask** | Changes the signal mask for the calling kernel thread. |
| **sleep** | Forces the calling kernel thread to wait on a specified channel. |

| Item | Description |
| --- | --- |
| **thread_create** | Creates a new kernel-only thread in the calling process. |
| **thread_self** | Returns the caller's kernel thread ID. |
| **thread_setsched** | Sets kernel thread scheduling parameters. |
| **thread_terminate** | Terminates the calling kernel thread. |
| **ue_proc_check** | Determines if a process is critical to the system. |
| **uprintf** | Submits a request to print a message to the controlling terminal of a process. |

# RAS Kernel Services

The Reliability, Availability, and Serviceability (RAS) kernel services are used to record the occurrence of hardware or software failures and to capture data about these failures.

The **panic** kernel service is called when a catastrophic failure occurs and the system can no longer operate. The **panic** service performs a system dump. The system dump captures data areas that are registered in the Master Dump Table. The kernel and kernel extensions use the **dmp_ctl** kernel service to add and delete entries in the Master Dump Table, and record dump routine failures.

The **errsave and errlast** kernel service is called to record an entry in the system error log when a hardware or software failure is detected.

The **trcgenk** and **trcgenkt** kernel services are used along with the **trchook** subroutine to record selected system events in the event-tracing facility.

The **ras_register** and **ras_unregister** kernel services register and unregister RAS handlers for a specific component. These handlers are called by the kernel when the system needs to communicate various RAS commands to each component.

The **register_HA_handler** and **unregister_HA_handler** kernel services are used to register high availability event handlers for kernel extensions that need to be aware of events such as processor deallocation.

One of the RAS features is a service that monitors for excessive periods of interrupt disablement on a processor, and logs these events to the error log. The **disablement_checking_suspend** and **disablement_checking_resume** services exempt a code segment from this detection.

## List of RAS Kernel Services

The RAS kernel services record the hardware and software failure occurrences and details.

The recorded information about the occurrence of hardware or software failures can be examined using the **errpt** or **trcrpt** commands.

| Item | Description |
| --- | --- |
| **disablement_checking_resume** | Restarts the ability to have excessive periods of interrupt disablement be detected |
| **disablement_checking_suspend** | Is exempt from detection of excessive periods of interrupt disablement. |
| **dmp_ctl** | Adds and removes entries to the master dump table. |
| **errsave and errlast** | Allows the kernel and kernel extensions to write to the error log. |
| **panic** | Crashes the system. |
| **ras_register** | Registers a component to perform and be notified of various RAS activities. |

| Item | Description |
| --- | --- |
| **ras_unregister** | Unregisters a component from taking part in RAS activities. |
| **register_HA_handler** | Registers a High Availability Event Handler |
| **trcgenk** | Records a trace event for a generic trace channel. |
| **trcgenkt** | Records a trace event, including a time stamp, for a generic trace channel. |
| **unregister_HA_handler** | Cancels the registration of a High Availability Event Handler |

## Security Kernel Services

The Security kernel services provide methods for controlling the auditing system and for determining the access rights to objects for the invoking process.

The following services are security kernel services:

| Item | Description |
| --- | --- |
| **suser** | Determines the privilege state of a process. |
| **audit_svcstart** | Initiates an audit record for a system call. |
| **audit_svcbcopy** | Appends event information to the current audit event buffer. |
| **audit_svcfinis** | Writes an audit record for a kernel service. |
| **crcopy** | Creates a copy of a security credentials structure. |
| **crdup** | Creates a copy of the current security credentials structure. |
| **credential macros** | Provide a means for accessing the user and group identifier fields within a credentials structure. |
| **crexport** | Copies an internal format credentials structure to an external format credentials structure. |
| **crfree** | Frees a security credentials structure. |
| **crget** | Allocates a new, uninitialized security credentials structure. |
| **crhold** | Increments the reference count of a security credentials structure. |
| **crref** | Increments the reference count of the current security credentials structure. |
| **crset** | Replaces the current security credentials structure. |
| **kcred_genpagvalue** | Generates a system-wide unique PAG value for a given PAG name (such as afs). |
| **kcred_getcap** | Copies a capability vector from a credentials structure. |
| **kcred_getgroups** | Copies the concurrent group set from a credentials structure. |
| **kcred_getpag** | Copies a process authentication group (PAG) ID from a credentials structure. |
| **kcred_getpag64** | Retrieves 64-bit PAG values from a process's credentials structure. |
| **kcred_getpagid** | Returns the process authentication group (PAG) identifier for a PAG name. |
| **kcred_getpagname** | Retrieves the name of a process authentication group (PAG). |
| **kcred_getpriv** | Copies a privilege vector from a credentials structure. |
| **kcred_setcap** | Copies a capabilities set into a credentials structure. |
| **kcred_setgroups** | Copies a concurrent group set into a credentials structure. |
| **kcred_setpag** | Copies a process authentication group ID into a credentials structure. |

| Item | Description |
|---|---|
| **kcred_setpag64** | Stores 64-bit PAG values in a process's credentials structure. |
| **kcred_setpagname** | Copies a process authentication group ID into a credentials structure. |
| **kcred_setpriv** | Copies a privilege vector into a credentials structure. |
| **TE_verify_reg** | Registers a callout handler for trusted execution file verification during `exec()`, kernel extension loads, and library load operations. |
| **TE_verify_unreg** | Unregister a previously registered callout handler for trusted execution. |

# Timer and Time-of-Day Kernel Services

The Timer and Time-of-Day kernel services provide kernel extensions with the ability to be notified when a period of time has passed.

The **tstart** service supports a very fine granularity of time. The **timeout** service is built on the **tstart** service and is provided for compatibility with earlier versions of the operating system. The **w_start** service provides a timer with less granularity, but much cheaper path-length overhead when starting a timer.

The Timer and Time-of-Day kernel services are divided into the following categories:

## Time-Of-Day Kernel Services

The time-of-day kernel services are used to display or set the current time or current status from timer-adjustment values.

| Item | Description |
|---|---|
| **curtime** | Reads the current time into a time structure. |
| **kgettickd** | Retrieves the current status of the systemwide time-of-day timer-adjustment values. |
| **ksettimer** | Sets the systemwide time-of-day timer. |
| **ksettickd** | Sets the current status of the systemwide timer-adjustment values. |

## Fine Granularity Timer Kernel Services

The fine granularity timer kernel services manage the timer requests.

| Item | Description |
|---|---|
| **delay** | Suspends the calling process for the specified number of timer ticks. |
| **talloc** | Allocates a timer request block before starting a timer request. |
| **tfree** | Deallocates a timer request block. |
| **tstart** | Submits a timer request. |
| **tstop** | Cancels a pending timer request. |

For more information about using the Fine Granularity Timer services, see “Using Fine Granularity Timer Services and Structures” on page 85.

## Timer Kernel Services for Compatibility

The timer kernel services schedules a function on a specified time.

| Item | Description |
|---|---|
| **timeout** | Schedules a function to be called after a specified interval. |
| **timeoutcf** | Allocates or deallocates callout table entries for use with the **timeout** kernel service. |

| Item | Description |
| --- | --- |
| **untimeout** | Cancels a pending timer request. |

### Watchdog Timer Kernel Services

A watchdog timer is a device or software that performs a specific operation after a certain period of time if the system or program stops its regular service and then it does not recover on its own.

The watchdog timer kernel services follow:

| Item | Description |
| --- | --- |
| **w_clear** | Removes a watchdog timer from the list of watchdog timers that are known to the kernel. |
| **w_init** | Registers a watchdog timer with the kernel. |
| **w_setattr** | Sets attributes for a watchdog timer. |
| **w_start** | Starts a watchdog timer. |
| **w_stop** | Stops a watchdog timer. |

## Using Fine Granularity Timer Services and Structures

The **talloc**, **tstart**, **tstop**, and **tfree** services provide fine-resolution timing functions.

The **talloc**, **tstart**, **tstop**, and **tfree** timer services should be used when the following conditions are required:

- Timing requests for less than one second
- Critical timing
- Absolute timing

A kernel extension uses the **talloc** service to allocate a timer request block (struct **trb**). The kernel extension initializes the fields in the **trb** structure to specify the timer completion function, when the function is to receive control, and how the function is to receive control. The **tstart** service is used to schedule the timer event. The timer completion handler is called when the specified time is reached. The address of the timer request block is passed to the completion handler function. The timer completion handler can initialize and reschedule the timer request block using the **tstart** service. The **tstop** service is used to cancel a pending timer request. The **tfree** service is used to return a previously allocated timer request block to the system. Be sure that the **trb** structure is not active when it is freed by the **tstop** service.

Use the **talloc** service to allocate a timer request block. Do not allocate your own memory to contain timer request blocks.

Do not access or modify any fields of the **trb** structure between the time when a timer request block is started by the **tstart** service and the time when the completion handler runs, or a call to the **tstop** service is completed against the timer request block.

If you use the **tstart** service in the completion handler to reschedule the timer request block that was passed to the completion handler, be sure to initialize the `t->timeout` field and the appropriate **T_INCTERVAL**, **T_ABSOLUTE**, and **T_LOWRES** bits in the `t->flags` field. Use an OR operation to set the `t->flags` bits because the AIX® kernel timer service is using other bits in the `t->flags` field when the completion handler is called.

The AIX® kernel timer service accesses the timer request block that was passed to the completion handler after the completion handler returned. A race condition might occur if a completion handler makes the timer request block that was passed to the completion handler available for the reuse by another processor. A system outage might occur if the other processor uses the **tstart** service to reschedule the timer request block before the AIX® kernel timer service has completed its access to the block. The other processor can avoid the system outage using the **tstop** service to ensure that the timer request block is stopped before using the **tstart** service to reschedule the timer request block.

**Note:**

1. If a timer request block is rescheduled anywhere other than its completion handler, use the **tstop** service to ensure that the timer request block is not scheduled or running in its completion handler. Failure to do so might result in a system outage.
2. Use the **tstop** service against the timer request block before calling the **tstart** service if a timer request block is restarted using the **tstart** service anywhere other than its completion handler. Failure to do so can result in system stops.
3. Always use the **tstop** service to cancel a timer request before calling the **tfree** service. This is necessary to ensure that the timer request block is not on a system timer queue, running the completion handler, or returning to the system from a completion handler when the timer request block is freed. Failure to do so might result in a system outage.

Do not use the **tfree** service in a timer completion handler to free the timer request block that was passed to the completion handler.

The Watchdog timer services can be used for noncritical times having a one-second resolution. The **timeout** service can be used for noncritical times having a clock-tick resolution.

## Timer Services Data Structures

The **trb** (timer request) structure is found in the **/sys/timer.h** file. The **itimerstruc_t** structure contains the second/nanosecond structure for time operations and is found in the **sys/time.h** file.

The **trb** structure contains a number of fields. Some of the fields are used for AIX® kernel processing. Others are intended for timer service users. A timer service user must set the following fields in the timer request block after calling the **talloc** service and before calling the **tstart** service.

| Fields | Description |
|---|---|
| `t->flags` | Specifies the flags. You must set either the **T_INCINTERVAL** flag or the **T_ABSOLUTE** flag in the field. You can also set the **T_LOWRES** flag if you want. The **T_LOWRES** flag causes the system to round the `t->timeout` value to the next timer timeout. The advantage of using the **T_LOWRES** flag is that it prevents an extra timer interrupt from being generated. The timeout is rounded to a larger value. The **T_LOWRES** flag never causes more than 10 ms to be added to the timeout value because the system maintains timers at an interval of 10 ms.<br><br>You can set either the **T_MOVE_OK** or the **T_LATE_OK** flag to allow the system to perform optimizations with the timer. If the **T_MOVE_OK** flag is set, the system might move the timer to a different processor from where it was originally started. The **T_MOVE_OK** flag should be set when the you do not have a dependency on which processor the timer expires. If the **T_LATE_OK** flag is set, the timer expiration might be delayed when the owning processor is in a sleep state. The **T_LATE_OK** flag should be set if the **T_MOVE_OK** flag is not applicable and you can tolerate an arbitrarily late expiration. If both flags are set, the **T_MOVE_OK** flag has precedence over the **T_LATE_OK** flag.<br><br>After the **T_MOVE_OK** or **T_LATE_OK** flag is set, both flags persist until explicitly cleared by the owner.<br><br>Both flags are optional and are provided to aid internal system optimization. |
| `t->timeout` | Specifies the timeout value. This field is used for both the interval timer and the absolute timer as indicated by the `t->flags` field. |
| `t->func` | Specifies the completion-handler function pointer. This function must remain in pinned memory as long as the function might be called. Use the **tstop** service to cancel any pending timer request blocks that can call the completion handler before unloading the kernel extension containing the function. |

| Fields | Description |
|---|---|
| `t->ipri` | Specifies the interrupt priority at which the completion handler is called. Completion handlers are called in the interrupt environment. The interrupt priority values are defined in the **/usr/include/sys/intr.h** file. The least favored interrupt priority for a timer completion handler is INTTIMER. The most favored interrupt priority is INTMAX. For more information about interrupt priorities, see "Understanding Interrupts" on page 50 in *AIX Version 6.1 Kernel Extensions and Device Support Programming Concepts*. |
| `t->t_union` | (optional) Passes data to the completion handler. |
| `t->id` | (optional) Specifies the ID of the process that schedules the timer request block. Device drivers can set the field to a value of -1. |

The `t->func`, `t->t_union`, `t->ipri`, and `t->id` fields persist across a call from **tstart** to the completion handler. Only the `t->timeout` and `t->flags` fields must be reset before a subsequent call to the **tstart** service.

Other fields in the **trb** structure do not need to be initialized between the **talloc** and **tstart** services.

The **itimerstruc_t t.it** value substructure is used to store time information for both absolute and incremental timers. The **T_ABSOLUTE** absolute request flag is defined in the **sys/timer.h** file.

### Coding the Timer Function

The `t->func` timer function is called on an interrupt level. Therefore, the code for this function must be in pinned storage and must follow conventions for interrupt handlers.

The `t->func` timer function should be declared as follows:

```
void func (t)
struct trb *t;
```

The argument to the **func** completion handler routine is the address of the **trb** structure, not the contents of the `t_union` field.

## Using Multiprocessor-Safe Timer Services

On a multiprocessor system, timer request blocks and watchdog timer structures could be accessed simultaneously by several processors.

The kernel services shown below potentially alter critical information in these blocks and structures, and therefore check whether it is safe to perform the requested service before proceeding:

| Item | Description |
|---|---|
| **tstop** | Cancels a pending timer request. |
| **w_clear** | Removes a watchdog timer from the list of watchdog timers known to the kernel. |
| **w_init** | Registers a watchdog timer with the kernel. |

If the requested service cannot be performed, the kernel service returns an error value.

In order to be multiprocessor safe, the caller must check the value returned by these kernel services. If the service was not successful, the caller must take an appropriate action, for example, retrying in a loop. If the caller holds a device driver lock, it should release and then reacquire the lock within this loop in order to avoid deadlock.

Drivers which were written for uniprocessor systems do not check the return values of these kernel services and are not multiprocessor-safe. Such drivers can still run as funnelled device drivers.

## Virtual File System (VFS) Kernel Services

The Virtual File System (VFS) kernel services are provided as fundamental building blocks for use when writing a virtual file system.

These services present a standard interface for such functions as configuring file systems, creating and freeing v-nodes, and looking up path names.

Most functions involved in the writing of a file system are specific to that file system type. But a limited number of functions must be performed in a consistent manner across the various file system types to enable the logical file system to operate independently of the file system type.

The VFS kernel services are:

| Item | Description |
|---|---|
| **common_reclock** | Implements a generic interface to the record locking functions. |
| **fidtovp** | Maps a file system structure to a file ID. |
| **gfsadd** | Adds a file system type to the **gfs** table. |
| **gfsdel** | Removes a file system type from the **gfs** table. |
| **vfs_hold** | Holds a **vfs** structure and increments the structure's use count. |
| **vfs_unhold** | Releases a **vfs** structure and decrements the structure's use count. |
| **vfsrele** | Releases all resources associated with a virtual file system. |
| **vfs_search** | Searches the vfs list. |
| **vn_free** | Frees a v-node previously allocated by the **vn_get** kernel service. |
| **vn_get** | Allocates a virtual node and associates it with the designated virtual file system. |
| **lookupvp** | Retrieves the v-node that corresponds to the named path. |

# Asynchronous I/O Subsystem

*Synchronous input/output (I/O)* occurs while you wait. Applications processing cannot continue until the I/O operation is complete. In contrast, *asynchronous I/O (AIO)* operations run in the background and do not block user applications. This improves performance, because I/O operations and application processing can run simultaneously.

Many applications, such as databases and file servers, take advantage of the ability to overlap processing and I/O. These AIO operations use various kinds of devices and files. Additionally, multiple AIO operations can run at the same time on one or more devices or files. Using AIO usually improves I/O throughput for these types of applications. The actual performance, however, depends partly on the number of concurrent I/O requests that the application can issue at one time. When the AIO fast path is not used, the performance also depends on how many AIO server processes that handle the I/O requests are running. For more information about the fast path, see “Identifying the number of AIO servers used currently” on page 90.

Each AIO request has a corresponding control block in the application's address space. When an AIO request is made, a handle is established in the control block. This handle is used to retrieve the status and the return values of the request.

Applications use the **aio_read** and **aio_write** subroutines to perform the I/O. Control returns to the application from the subroutine, as soon as the request has been queued. The application can then continue processing while the disk operation is being performed.

A kernel process (kproc), called an AIO server (AIOS), is in charge of each request from the time it is taken off the queue until it completes. The number of servers limits the number of disk I/O operations that can be in progress in the system simultaneously.

The default value of the *minservers* tunable is 3, and that of the *maxservers* tunable is 30. In systems that seldom run applications that use AIO, this is usually adequate. For environments with many disk drives and key applications that use AIO, the defaults might be too low. The result of a deficiency of servers is that disk I/O seems much slower than it should be. Not only do requests spend inordinate lengths of time in the queue, but the low ratio of servers to disk drives means that the seek-optimization algorithms have too few requests to work with for each drive.

**Note:** AIO does not work if the control block or buffer is created using mmap (mapping segments).

There are two AIO subsystems. The original AIX® AIO, now called LEGACY AIO, has the same function names as the Portable Operating System Interface (POSIX) compliant POSIX AIO. The major differences between the two involve different parameter passing. Both subsystems are defined in the `/usr/include/sys/aio.h` file. The **_AIO_AIX_SOURCE** macro is used to distinguish between the two versions.

**Note:** The **_AIO_AIX_SOURCE** macro used in the **/usr/include/sys/aio.h** file must be defined when using this file to compile an AIO application with the LEGACY AIO function definitions. The default compilation using the **aio.h** file is for an application with the new POSIX AIO definitions. To use the LEGACY AIO function definitions do the following in the source file:

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

or when compiling on the command line, type the following:

```
xlc ... -D_AIO_AIX_SOURCE ... classic_aio_program.c
```

For each AIO function there is a legacy and a POSIX definition. LEGACY AIO has an additional **aio_nwait** function, which although not a part of POSIX definitions, has been included in POSIX AIO to help those who want to port from LEGACY to POSIX definitions. POSIX AIO has an additional **aio_fsync** function, which is not included in LEGACY AIO. For a list of these functions, see “Asynchronous I/O Subroutines” on page 93.

**Related information**

System Management Interface Tool (SMIT)

## Understanding AIO

Using the **vmstat** command with an interval and count value, you can determine if the processor is idle waiting for disk I/O. The `wa` column details the percentage of time the processor was idle with pending local disk I/O.

If there is at least one outstanding I/O to a local disk when the wait process is running, the time is classified as waiting for I/O. Unless AIO is being used by the process, an I/O request to disk causes the calling process to block (or sleep) until the request is complete. After a process's I/O request completes, it is placed on the run queue.

A `wa` value consistently over 25 percent might indicate that the disk subsystem is not balanced properly, or it might be the result of a disk-intensive workload.

**Note:** AIO does not relieve an overly busy disk drive. Using the **iostat** command with an interval and count value, you can determine if any disks are overly busy. Monitor the `%tm_act` column for each disk drive on the system. On some systems, a `%tm_act` of 35.0 or higher for one disk can cause noticeably slower performance. The relief for this case could be to move data from more busy to less busy disks, but simply having AIO does not relieve an overly busy disk problem.

## SMP Systems

For SMP systems, the `us, sy, id` and `wa` columns are only averages over all processors. But remember that the I/O wait statistic per processor is not really a processor-specific statistic; it is a global statistic. An I/O wait is distinguished from idle time only by the state of a pending I/O.

If there is any pending disk I/O, and the processor is not busy, then it is an I/O wait time. Disk I/O is not tracked by processors, so when there is any I/O wait, all processors get charged (assuming they are all equally idle).

## Identifying the number of AIO servers used currently

This section provides information to determine how many POSIX and LEGACY AIO Servers are currently running.

To determine how many POSIX AIO Servers are currently running, type the following information on the command line:

```
pstat -a | grep posix_aioserver | wc -l
```

**Requirement:** You must run this command as the root user.

To determine how many LEGACY AIO Servers are currently running, type the following information on the command line:

```
pstat -a | egrep ' aioserver' | wc -l
```

**Requirement:** You must run this command as the root user.

A timeout value, specified by the *server_inactivity* tunable, causes a server to exit if the server is idle without serving an AIO request. If, when the server exits, the total number of servers falls below the value of the *minservers* tunable multiplied by the number of processors, the server becomes idle, waiting for an AIO request to serve. By reducing the number of idle processes that are not being used to serve AIO requests, overall system performance is enhanced.

If the data being accessed asynchronously is located in a Journaled File System (JFS), all I/O is routed through the AIOS kprocs.

If the data being accessed asynchronously is located on a raw logical volume or the Enhanced Journaled File System (JFS2) in conjunction with concurrent I/O (CIO), the I/O is routed using a fast path and does not go through the AIOS kprocs. In that case the number of servers that are running is not relevant.

However, if you want to confirm that an application that uses raw logic volumes or JFS2 with CIO is taking advantage of AIO, you can disable the fast path option using the **ioo** command. When this option is disabled, even raw logical volume and JFS2 I/O are forced through the AIOS kprocs. At that point, the **pstat** command listed in the preceding discussion works. Do not run the system with this option disabled for any length of time.

You can use the AIO fast path for files that are opened with CIO on the JFS2 file system.

## Identifying the number of AIO servers

You can set the value for the maximum number of servers in a number of ways.

- Limit the maximum number of servers to a number equal to ten times the number of disks that are to be used concurrently.
- Set the maximum number of servers to a high number (for example, 100), depending on the system and the number of processors.
  - Change the *server_inactivity* tunable value to 600 (the default value is 300).
  - Monitor the number of servers every 10 minutes (600 seconds) throughout the course of a normal workload. If the number of servers is constantly at the aggregate value for the *maxservers* tunable

(the *maxservers* tunable value times the number of processors) and there is unused processor and IO bandwidth, consider increasing the *maxservers* tunable value. If you find that at times the system is not performing as expected, consider lowering the *maxservers* tunable value.

The goal is to find the performance balance between processor usage and IO bandwidth.

- Take statistics using **vmstat -s** before any high I/O activity begins, and again at the end. Check the field `iodone`. From this you can determine how many physical I/Os are being handled in a given wall clock period. Then increase the maximum number of servers and see if you can get more activity or event completions (iodones) in the same time period.

  In general, consider changing the *minservers* tunable value only when an application will be issuing a high number of I/Os that is beyond the *server_inactivity* timeout value (thus causing a high number of AIOS kprocs to be created in bursts). To help smooth this condition, either increase the *minservers* value to keep the required number of AIOS kprocs active, or increase the *server_inactivity* tunable value so that the system will naturally keep them active if the application keeps issuing requests within the timeout window.

**Requirement:**

- You must set the *minservers* tunable value at a level so that optimal performance can be obtained across an average workload. You do not need to restart the system to effect a change to the *minservers* or *maxservers* tunable.
- The value of the *minservers* tunable cannot exceed that of the *maxservers* tunable.

### Tunable Values for Asynchronous I/O

The *aio_minreqs*, *aio_minservers*, *aio_maxservers*, and *aio_server_inactivity* nonrestricted tunables are for the LEGACY AIO subsystem; and the *posix_aio_minreqs*, *posix_aio_minservers*, *posix_aio_maxservers*, *posix_aio_server_inactivity* nonrestricted tunables are for the POSIX AIO subsystem.

For more information about each tunable, see the **ioo** command.

## Functions of Asynchronous I/O

This overview describes some of the basic concepts and functions of asynchronous I/O.

### Large File-Enabled Asynchronous I/O

The fundamental data structure associated with all AIO operations is **struct aiocb**. Within this structure is the `aio_offset` field, which is used to specify the offset for an I/O operation.

Due to the signed 32-bit definition of `aio_offset`, the default AIO interfaces are limited to an offset of 2Gb minus 1. To overcome this limitation, a new AIO control block with a signed 64-bit offset field and a new set of AIO interfaces has been defined. These 64-bit definitions end with "64."

The large offset-enabled AIO interfaces are available under the _LARGE_FILES compilation environment and under the _LARGE_FILE_API programming environment. For further information, see Writing Programs That Access Large Files in *AIX Version 7.1 General Programming Concepts: Writing and Debugging Programs*.

Under the _LARGE_FILES compilation environment, AIO applications written to the default interfaces interpret the following redefinitions:

| Item | Redefined To Be | Header File |
|---|---|---|
| struct aiocb | struct aiocb64 | sys/aio.h |
| aio_read() | aio_read64() | sys/aio.h |
| aio_write() | aio_write64() | sys/aio.h |
| aio_cancel() | aio_cancel64() | sys/aio.h |
| aio_suspend() | aio_suspend64() | sys/aio.h |

| Item | Redefined To Be | Header File |
|---|---|---|
| aio_listio() | aio_listio64() | sys/aio.h |
| aio_return() | aio_return64() | sys/aio.h |
| aio_error() | aio_error64() | sys/aio.h |

For information on using the _LARGE_FILES environment, see Porting Applications to the Large File Environment in *AIX Version 7.1 General Programming Concepts: Writing and Debugging Programs*.

In the _LARGE_FILE_API environment, the 64-bit application programming interfaces (APIs) are visible. This environment requires recoding of applications to the new 64-bit API name. For further information on using the _LARGE_FILE_API environment, see Using the 64-Bit File System Subroutines in *AIX Version 7.1 General Programming Concepts: Writing and Debugging Programs*.

## Nonblocking I/O

After issuing an I/O request, the user application can proceed without being blocked while the I/O operation is in progress. The I/O operation occurs while the application is running.

Specifically, when the application issues an I/O request, the request is queued. The application can then resume running before the I/O operation starts.

To manage AIO, each AIO request has a corresponding control block in the application's address space. This control block contains the control and status information for the request. It can be used again when the I/O operation finishes.

## Notification of I/O Completion

After issuing an AIO request, the user application can determine when and how the I/O operation finishes.

This information is provided in three ways:

### *Polling the Status of the I/O Operation*

The application can periodically poll the status of the I/O operation. The status of each I/O operation is provided in the application's address space in the control block associated with each request.

Portable applications can retrieve the status by using the **aio_error** subroutine. The **aio_suspend** subroutine suspends the calling process until one or more AIO requests finish.

### *Asynchronously Notifying the Application When the I/O Operation Completes*

Asynchronously notifying the I/O completion is done by signals.

Specifically, an application can request that a **SIGIO** signal be delivered when the I/O operation is complete. To do this, the application sets a flag in the control block at the time it issues the I/O request. If several requests are issued, the application can poll the status of the requests to determine which ones completed.

### *Blocking the Application until the I/O Operation Is Complete*

The third way to determine whether an I/O operation is complete is to let the calling process become blocked and wait until at least one of the I/O requests it is waiting for is complete.

This method is similar to synchronous style I/O. It is useful for applications that, after performing some processing, need to wait for I/O completion before proceeding.

## Cancellation of I/O Requests

Some I/O requests can be canceled. Cancellation is not guaranteed and might succeed or not depending upon the state of the individual request.

If a request is in the queue and the I/O operations have not yet started, the request can be canceled. Typically, a request can no longer be canceled after the actual I/O operation begins.

## Asynchronous I/O Subroutines

The 64-bit subroutines that are listed in this section along with their purpose are provided for performing AIO.

| Subroutine | Purpose |
|---|---|
| **aio_cancel or aio_cancel64** | Cancels one or more outstanding AIO requests. |
| **aio_error or aio_error64** | Retrieves the error status of an AIO request. |
| **aio_fsync** | Synchronizes asynchronous files. |
| **lio_listio or lio_listio64** | Initiates a list of AIO requests with a single call. |
| **aio_nwait** | Suspends the calling process until *n* AIO requests are completed. |
| **aio_read or aio_read64** | Reads asynchronously from a file. |
| **aio_return or aio_return64** | Retrieves the return status of an AIO request. |
| **aio_suspend or aio_suspend64** | Suspends the calling process until one or more AIO requests finishes. |
| **aio_write or aio_write64** | Writes asynchronously to a file. |

### Order and Priority of Asynchronous I/O Calls

An application can issue several AIO requests on the same file or device.

However, because the I/O operations are performed asynchronously, the order in which they are handled might not be the order in which the I/O calls are made. The application must enforce ordering of its own I/O requests if ordering is required.

**Note:** Priority among the I/O requests is available only for POSIX AIO.

For files that support **seek** operations, seeking can be done as part of the asynchronous read or write operations. The `whence` and `offset` fields are provided in the control block of the request to set the *seek* parameters. The *seek* pointer is updated when the asynchronous read or write call returns.

## Subroutines Affected by Asynchronous I/O

The existing subroutines that are listed in this section are affected by AIO.

- The **close** subroutine
- The **exit** subroutine
- The **exec** subroutine
- The **fork** subroutine

If the application closes a file, or calls the **_exit** or **exec** subroutines while it has some outstanding I/O requests, the requests are canceled. If they cannot be canceled, the application is blocked until the requests finish. When a process calls the **fork** subroutine, its AIO is not inherited by the child process.

One fundamental limitation in AIO is page hiding. When an unbuffered (raw) AIO is issued, the page that contains the user buffer is hidden during the actual I/O operation. This ensures cache consistency. However, the application can access the memory locations that fall within the same page as the user buffer. This can cause the application to block as a result of a page fault. To alleviate this, allocate page aligned buffers and do not touch the buffers until the I/O request using it finishes.

## 64-bit Enhancements

AIO has been enhanced to support 64-bit enabled applications. On 64-bit platforms, both 32-bit and 64-bit AIO can occur simultaneously.

The **aiocb** structure, the fundamental data structure associated with all AIO operations, has changed. The element of this struct, **aio_return**, is now defined as `ssize_t`. Previously, it was defined as an `int`. AIO

supports large files by default. An application compiled in 64-bit mode can do AIO to a large file without any additional #define specifications or special opening of those files.

# LEGACY AIO Extended Functionality

The extended functionality supported by LEGACY AIO includes extended asynchronous I/O control block (AIOCB), I/O priorities and cache hints, and I/O completion ports.

## Extended AIOCB

LEGACY AIO supports functionality that is not available in POSIX AIO.

To extend the LEGACY AIOCB, the aio_reqprio and aio_fp fields are deprecated, and the following new fields are introduced:

| Field | Version |
|---|---|
| aio_version | All versions. |
| aio_priority | AIOCBX_VERS1 |
| aio_cache_hint | AIOCBX_VERS1 |
| aio_iocpfd | AIOCBX_VERS2 |

A new flag, AIO_EXTENDED, has also been added to the aio_flags field. If the AIO_EXTENDED flag is not set, LEGACY AIO completely ignores any new extended fields. If the AIO_EXTENDED flag is set within aio_flags, and the aio_version field contains a value greater than 0 and less than or equal to AIOCBX_VERSION, all extended fields with a version indicated in the preceding table that are less than or equal to the version number specified in the AIOCB are in force. Future extensions to the legacy AIOCB structure will use new version values and introduce new extended fields beyond what is currently defined within the AIOCB structure.

Except for the aio_version field, all extended fields are required to ignore a value of 0 (zero). A user of any extended field must ensure that all other unused extended fields are initialized to zero. Use either the bzero or memset function on the entire AIOCB structure prior to setting any field in the structure.

## I/O Priorities and Cache Hints

To use I/O priorities and cache hints with AIO, set the AIO_EXTENDED flag in the aio_flags field and the aio_version field to a value of AIOCBX_VERS1 or greater. All other extended fields that are defined must be set to 0 if they are not used.

The following fields are used with this extended functionality:

| Field | Version |
|---|---|
| aio_priority | AIOCBX_VERS1 |
| aio_cache_hint | AIOCBX_VERS1 |

The aio_priority and aio_cache_hint values take effect only on a 64-bit kernel under the following conditions:

- The file descriptor being operated on by AIO belongs to the raw character interface of an LVM logical volume.
- The LVM logical volume resides on a device that supports I/O priorities and cache hints.

The aio_read, aio_read64, aio_write, aio_write64, lio_listio, and lio_listio64 interfaces are all compatible with an extended AIOCB. Other interfaces (such as aio_cancel) ignore the extended fields.

The valid values for aio_priority and aio_cache_hint are described in the sys/extendio.h file. The aio_priority must be either IOPRIORITY_UNSET (0) or a value from 1 to 15. Lower I/O priority values are considered to be more important than higher values. For example, a value of 1 is considered highest priority and a value of 15 is considered lowest priority. The aio_cache_hint must be either CH_AGE_OUT_FAST or CH_PAGE_WRITE. These cache hint values are mutually exclusive. If CH_AGE_OUT_FAST is set, the I/O buffer can be aged out quickly from the storage device buffer cache. This is useful in situations where the application is already caching the I/O buffer and redundant caching within the storage layer can be avoided. If CH_PAGE_WRITE is set, the I/O buffer is written only to the storage device cache and not to the disk.

## Using I/O Completion Ports with AIO Requests

To use I/O completion ports (IOCP) with AIO requests, set the AIO_EXTENDED flag in the aio_flags field and the aio_version field to a value of AIOCBX_VERS2 or higher. All other extended fields defined must be set to 0 if they are not used.

The following fields are used with this extended functionality:

| Field | Version |
|---|---|
| aio_iocpfd | AIOCBX_VERS2 |

A limitation of the AIO interface that is used in a threaded environment is that aio_nwait() collects completed I/O requests for ALL threads in the same process. In other words, one thread collects completed I/O requests that are submitted by another thread. Another problem is that multiple threads cannot invoke the collection routines (such as aio_nwait()) at the same time. If one thread issues aio_nwait() while another thread is calling it, the second aio_nwait() returns EBUSY. This limitation can affect I/O performance when many I/Os must run at the same time and a single thread cannot run fast enough to collect all the completed I/Os.

Using I/O completion ports with AIO requests provides the capability for an application to capture results of various AIO operations on a per-thread basis in a multithreaded environment. This functionality provides threads with a method of receiving completion status for only the AIO requests initiated by the thread.

The IOCP subsystem only provides completion status by generating completion packets for AIO requests. The I/O cannot be submitted for regular files through IOCP.

The current behavior of AIO remains unchanged. An application is free to use any existing AIO interfaces in combination with I/O completion ports. The application is responsible for "harvesting" completion packets for any noncanceled AIO requests that it has associated with a completion port.

The application must associate a file with a completion port using the CreateIoCompletionPort IOCP routine. The file can be associated with multiple completion ports, and a completion port can have multiple files associated with it. When making the association, the application must use an application-defined *CompletionKey* to differentiate between AIO completion packets and socket completion packets. The application can use different *CompletionKeys* to differentiate among individual files (or in any other manner) as necessary.

The application must also associate AIO requests with the same completion port as the corresponding file. It does this by initializing the aio_iocpfd of the AIOCB with the file descriptor of the completion port. An AIOCB can be associated with only one completion port, but a completion port can have multiple AIOCBs associated with it. The association between a completion port and an AIOCB must be done *before* the request is made. This is accomplished using an AIO routine, such as aio_write, aio_read, or lio_listio. If the value in the aio_iocpfd field is not a valid completion port file descriptor, the attempt to start the request fails and no I/O is performed.

An association must be made directly between a completion port and an AIOCB. For example, if you want to call lio_listio(), each AIOCB in the lio_listio chain must be associated individually prior to the call. It is not necessary to have all AIOCBs in the chain associated with a completion port.

After an association is made, it remains until the application explicitly clears it by using a value of 0 for the `aio_iocpfd` field, or the AIOCB is destroyed. A completion packet is created only when I/O completes for an AIOCB that has been associated with a completion port.

A summary of the steps that an application takes to use I/O completion ports with AIO requests is as follows:

1. Opens a regular file for I/O.
2. Calls the `CreateIoCompletionPort` routine to create an I/O completion port (IOCP), using the file descriptor for the regular file and an application-defined *CompletionKey*, which is used to differentiate AIO requests from socket I/O. The `CreateIoCompletionPort` function returns an IOCP file descriptor that corresponds to the newly created IOCP.
3. Allocates and clears (using the `bzero` function) an AIO control block. Indicates that I/O completion ports are to be used with AIO requests by setting the `AIO_EXTENDED` flag of the AIOCB's `aio_flags` field. Also sets the `aio_version` field to a value of `AIOCBX_VERS2` or higher.
4. Associates the AIO request with the IOCP by initializing the `aio_iocpfd` field in the AIOCB to contain the IOCP file descriptor returned by the `CreateIoCompletionPort` routine.
5. Starts the AIO request using existing AIO interfaces. Multiple requests can be started using the `lio_listio` interface.
6. Calls the `GetQueuedCompletionStatus` function with the IOCP file descriptor to collect the results of the completed AIO requests on a particular IOCP. The application provides the address of a pointer in the LPOVERLAPPED argument to `GetQueuedCompletionStatus`, so the corresponding AIOCB pointer can be returned. Details of the AIO request can be determined by examining the returned AIOCB.
7. After all I/O is complete, the application is responsible for closing all file descriptors.

## Device Configuration Subsystem

Devices are usually pieces of equipment that attach to a computer. Devices include printers, adapters, and disk drives. Additionally, devices are special files that can handle device-related tasks.

System users cannot operate devices until device configuration occurs. To configure devices, the Device Configuration Subsystem is available.

Review the following commands and subroutines before you configure a device:

- **getattr** subroutine
- **ioctl** subroutine
- **odm_run_method** subroutine
- **putattr** subroutine
- **sysconfig** subroutine
- **cfgmgr** command
- **chdev** command
- **mkdev** command
- **rmdev** command
- SYS_CFGDD **sysconfig** operation
- **ddconfig** device driver entry point

**Related information**

Understanding System Boot Processing
Special Files
Initial Printer Configuration
Machine Device Driver
Loading a Device Driver

## Scope of Device Configuration Support

The term *device* has a wider range of meaning in this operating system than in traditional operating systems. Traditionally, *devices* refers to hardware components such as disk drives, tape drives, printers, and keyboards. Pseudo-devices, such as the console, **error** special file, and **null** special file, are also included in this category.

However, in this operating system, all of these devices are referred to as *kernel devices*, which have device drivers and are known to the system by major and minor numbers.

Also, in this operating system, hardware components such as buses, adapters, and enclosures (including racks, drawers, and expansion boxes) are considered devices.

## Device Configuration Subsystem Overview

Devices are organized hierarchically within the system. This organization requires lower-level device dependence on upper-level devices in child-parent relationships. The system device (sys0) is the highest-level device in the system node, which consists of all physical devices in the system.

Each device is classified into functional classes, functional subclasses and device types (for example, printer *class*, parallel *subclass*, 4201 Proprinter *type*). These classifications are maintained in the device configuration databases with all other device information.

The Device Configuration Subsystem consists of:

| Item | Description |
|---|---|
| **High-level Commands** | Maintain (add, delete, view, change) configured devices within the system. These commands manage all of the configuration functions and are performed by invoking the appropriate device methods for the device being configured. These commands call device methods and low-level commands.<br><br>The system uses the high-level **Configuration Manager** (**cfgmgr**) command used to invoke automatic device configurations through system boot phases and the user can invoke the command during system run time. *Configuration rules* govern the **cfgmgr** command. |

| Item | Description |
| --- | --- |
| **Device Methods** | Define, configure, change, unconfigure, and undefine devices. The device methods are used to identify or change the device *states* (operational modes). |
| **Database** | Maintains data through the *ODM* (Object Data Manager) by object classes. Predefined Device Objects contain configuration data for all devices that can possibly be used by the system. Customized Device Objects contain data for *device instances* that are actually in use by the system. |

# General Structure of the Device Configuration Subsystem

Data that is used by the three levels is maintained in the *Configuration database*.

The *Configuration database* is managed as object classes by the Object Data Manager (ODM). All information relevant to support the device configuration process is stored in the configuration database.

The system cannot use any device unless it is configured.

The database has two components: the Predefined database and the Customized database. The *Predefined database* contains configuration data for all devices that could possibly be supported by the system. The *Customized database* contains configuration data for the devices actually defined or configured in that particular system.

The *Configuration manager* (**cfgmgr** command) performs the configuration of a system's devices automatically when the system is booted. This high-level program can also be invoked through the system keyboard to perform automatic device configuration. The configuration manager command configures devices as specified by *Configuration rules*.

The Device Configuration Subsystem can be viewed from the following different levels:

## High-Level Perspective

From a high-level, user-oriented perspective, device configuration comprises the basic tasks that are listed in this section.

- Adding a device to the system
- Deleting a device from the system
- Changing the attributes of a device
- Showing information about a device

From a high-level, system-oriented perspective, device configuration provides the basic task of automatic device configuration: running the configuration manager program.

A set of high-level commands accomplish all of these tasks during run time: **chdev**, **mkdev**, **lsattr**, **lsconn**, **lsdev**, **lsparent**, **rmdev**, and **cfgmgr**. High-level commands can invoke device methods and low-level commands.

## Device Method Level

Beneath the high-level commands (including the **cfgmgr** Configuration Manager program) is a set of functions called *device methods*. These methods perform well-defined configuration steps, including these five functions:

- Defining a device in the configuration database
- Configuring a device to make it available
- Changing a device to make a change in its characteristics
- Unconfiguring a device to make it unavailable
- Undefining a device from the configuration database

"Understanding Device States" on page 103 discusses possible device states and how the various methods affect device state changes.

The high-level device commands (including **cfgmgr**) can use the device methods. These methods insulate high-level configuration programs from kernel-specific, hardware-specific, and device-specific configuration steps. Device methods can invoke low-level commands.

### Low-Level Perspective

Beneath the device methods is a set of low-level library routines.

These library routines can be directly called by device methods as well as by high-level configuration programs.

## Device Configuration Database Overview

The Configuration database is an object-oriented database. The Object Data Manager (ODM) provides facilities for accessing and manipulating it through object classes.

The following databases are used in the configuration process:

| Item | Description |
|---|---|
| **Predefined database** | Contains information about all possible types of devices that can be defined for the system. |
| **Customized database** | Describes all devices currently defined for use in the system. Items are referred to as *device instances*. |

ODM Device Configuration Object Classes provides access to the object classes that make up the Predefined and Customized databases.

Devices must be defined in the database for the system to make use of them. For a device to be in the Defined state, the Configuration database must contain a complete description of it. This information includes items such as the device driver name, the device major and minor numbers, the device method names, the device attributes, connection information, and location information.

## Basic Device Configuration Procedures Overview

At system boot time, **cfgmgr**) is automatically invoked to configure all devices detected as well as any device whose device information is stored in the Configuration database. At run time, you can configure a specific device by directly invoking (or indirectly invoking through a usability interface layer) high-level device commands.

High-level device commands invoke methods and allow the user to add, delete, show, and change devices and their associated attributes.

When a specific device is defined through its define method, the information from the Predefined database for that type of device is used to create the information describing the specific device instance. This specific device instance information is then stored in the Customized database.

The process of configuring a device is often highly device-specific. The configure method for a kernel device must:

- Load the device's driver into the kernel.
- Pass the device dependent structure (DDS) describing the device instance to the driver. For more information on DDS, see "Device Dependent Structure (DDS) Overview" on page 107.
- Create a special file for the device in the **/dev** directory. For more information, see Special Files.

Many devices do not have device drivers. For this type of device the configured state is not as meaningful. However, it still has a Configure method that simply marks the device as configured or performs more complex operations to determine if there are any devices attached to it.

The configuration process requires that a device be defined or configured before a device attached to it can be defined or configured. At system boot time, the Configuration Manager first configures the system device. The remaining devices are configured by traversing down the parent-child connections layer by layer. The Configuration Manager then configures any pseudo-devices that need to be configured.

# Device Configuration Manager Overview

The Configuration Manager is a rule-driven program that automatically configures devices in the system during system boot and run time. When the Configuration Manager is invoked, it reads rules from the Configuration Rules object class and performs the indicated actions.

For more information on Configuration Rules, see Configuration Rules (Config_Rules) Object Class.

Devices in the system are organized in clusters of tree structures known as *nodes*. Each tree is a logical subsystem by itself. For example, the system node consists of all the physical devices in the system. The top of the node is the system device. Below the bus and connected to it are the adapters. The bottom of the hierarchy contains devices to which no other devices are connected. Most pseudo-devices, including low -function terminal (LFT) and pseudo-terminal (pty) devices, are organized as separate tree structures or nodes.

## Devices Graph

Devices, or nodes, in the system are organized in clusters of tree structures or device graphs.

See “Understanding Device Dependencies and Child Devices” on page 105 for more information.

## Configuration Rules

Each rule in the Configuration Rules (Config_Rules) object class specifies a program name that the Configuration Manager must execute. These programs are typically the configuration programs for the devices at the top of the nodes. When these programs are invoked, the names of the next lower-level devices that need to be configured are returned.

The Configuration Manager configures the next lower-level devices by invoking the configuration methods for those devices. In turn, those configuration methods return a list of to-be-configured device names. The process is repeated until no more device names are returned. As a result, all devices in the same node are configured in transverse order. The following are different types of rules:

- Phase 1
- Phase 2
- Service

The system boot process is divided into two phases. In each phase, the Configuration Manager is invoked. During phase 1, the Configuration Manager is called with a **-f** flag, which specifies that *phase = 1* rules are to be executed. This results in the configuration of base devices into the system, so that the root file system can be used. During phase 2, the Configuration Manager is called with a **-s** flag, which specifies that *phase = 2* rules are to be executed. This results in the configuration of the rest of the devices into the system.

For more information on the booting process, see Understanding System Boot Processing in *Operating system and device management*.

The Configuration Manager invokes the programs in the order specified by the sequence value in the rule. In general, the lower the sequence number within a given phase, the higher the priority. Thus, a rule with a 2 sequence number is executed before a rule with a sequence number of 5. An exception is made for 0 sequence numbers, which indicate a don't-care condition. Any rule with a sequence number of 0 is executed last. The Configuration Rules (Config_Rules) object class provides an example of this process.

If device names are returned from the program invoked, the Configuration Manager finishes traversing the node tree before it invokes the next program. Note that some program names might not be associated with any devices, but they must be included to configure the system.

### Invoking the Configuration Manager

During system boot time, the Configuration Manager is run in two phases. In phase 1, it configures the base devices needed to successfully start the system. These devices include the root volume group, which permits the configuration database to be read in from the root file system.

In phase 2, the Configuration Manager configures the remaining devices using the configuration database from the root file system. During this phase, different rules are used, depending on whether the system was booted in normal mode or in service mode. If the system is booted in service mode, the rules for service mode are used. Otherwise, the phase 2 rules are used.

The Configuration Manager can also be invoked during run time to configure all the detectable devices that might have been turned off at system boot or added after the system boot. In this case, the Configuration Manager uses the phase 2 rules.

## Device Classes, Subclasses, and Types Overview

To manage the wide variety of devices it supports more easily, the operating system classifies them hierarchically. One advantage of this arrangement is that device methods and high-level commands can operate against a whole set of similar devices.

Devices are categorized into the following main groups:

- Functional classes
- Functional subclasses
- Device types

Devices are organized into a set of *functional classes* at the highest level. From a user's point of view, all devices belonging to the same class perform the same functions. For example, all printer devices basically perform the same function of generating printed output.

However, devices within a class can have different interfaces. A class can therefore be partitioned into a set of *functional subclasses* in which devices belonging to the same subclass have similar interfaces. For example, serial printers and parallel printers form two subclasses of printer devices.

Finally, a device subclass is a collection of *device types*. All devices belonging to the same device type share the same manufacturer's model name and number. For example, 3812-2 (model 2 Pageprinter) and 4201 (Proprinter II) printers represent two types of printers.

Devices of the same device type can be managed by different drivers if the type belongs to more than one subclass. For example, the 4201 printer belongs to both the serial interface and parallel interface subclasses of the printer class, although there are different drivers for the two interfaces. However, a device of a particular class, subclass, and type can be managed by only one device driver.

Devices in the system are organized in clusters of tree structures known as *nodes*. For example, the system node consists of all the physical devices in the system. At the top of the node is the system device. Below the bus and connected to it are the adapters. The bottom of the hierarchy contains the devices to which no other devices are connected. Most pseudo-devices, including LFT and PTY, are organized as separate nodes.

## Writing a Device Method

*Device methods* are programs associated with a device that perform basic device configuration operations. These operations consist of defining, undefining, configuring, unconfiguring, and reconfiguring a device. Some devices also use optional start and stop operations.

The following are the basic device methods:

| Item | Description |
|---|---|
| **Define** | Creates a device instance in the Customized database. |

| Item | Description |
| --- | --- |
| **Configure** | Configures a device instance already represented in the Customized database. This method is responsible for making a device available for use in the system. |
| **Change** | Reconfigures a device by allowing device characteristics or attributes to be changed. |
| **Unconfigure** | Makes a configured device unavailable for use in the system. The device instance remains in the Customized database but must be reconfigured before it can be used. |
| **Undefine** | Deletes a device instance from the Customized database. |

### Invoking Methods

One device method can invoke another device method.

For instance, a Configure method for a device may need to invoke the Define method for child devices. The Change method can invoke the Unconfigure and Configure methods. To ensure proper operation, a method that invokes another method must always use the **odm_run_method** subroutine.

### Example Methods

See the **/usr/samples** directory for example device method source code.

These source code excerpts are provided for example purposes only. The examples do not function as written.

## Understanding Device Methods Interfaces

Device methods are not executed directly from the command line.

They are only invoked by the Configuration Manager at boot time or by the **cfgmgr**, **mkdev**, **chdev**, and **rmdev** configuration commands at run time. As a result, any device method you write should meet well-defined interfaces.

The parameters that are passed into the methods as well as the exit codes returned must both satisfy the requirements for each type of method. Additionally, some methods must write information to the **stdout** and **stderr** files.

These interfaces are defined for each of the device methods in the individual articles on writing each method.

To better understand how these interfaces work, one needs to understand, at least superficially, the flow of operations through the Configuration Manager and the run-time configuration commands.

### Configuration Manager

A node is a group of devices organized into a tree structure representing the various interconnections of the devices. The Node Configuration program is responsible for starting the configuration process for a node. It does this by querying the Customized database to see if the device at the top of the node is represented in the database. If so, the program writes the logical name of the device to the **stdout** file and then returns to the Configuration Manager.

The Configuration Manager begins by invoking a Node Configuration program listed in one of the rules in the Configuration Rules (Config_Rules) object class.

The Configuration Manager intercepts the Node Configuration program's **stdout** file to obtain the name of the device that was written. It then invokes the Configure method for that device. The device's Configure method performs the steps necessary to make the device available. If the device is not an intermediate one, the Configure method simply returns to the Configuration Manager. However, if the device is an intermediate device that has child devices, the Configure method must determine whether any of the child devices need to be configured. If so, the Configure method writes the names of all the child devices to be configured to the **stdout** file and then returns to the Configuration Manager.

The Configuration Manager intercepts the Configure method's **stdout** file to retrieve the names of the children. It then invokes, one at a time, the Configure methods for each child device. Each of these Configure methods operates as described for the parent device. For example, it might simply exit when complete, or write to its **stdout** file a list of additional device names to be configured and then exit. The Configuration Manager will continue to intercept the device names written to the **stdout** file and to invoke the Configure methods for those devices until the Configure methods for all the devices have been run and no more names are written to the **stdout** file.

### Run-Time Configuration Commands

User configuration commands invoke device methods during run time.

| Item | Description |
|---|---|
| **mkdev** | The **mkdev** command is invoked to define or configure, or define and configure, devices at run time. If just defining a device, the **mkdev** command invokes the Define method for the device. The Define method creates the customized device instance in the Customized Devices (CuDv) object class and writes the name assigned to the device to the **stdout** file. The **mkdev** command intercepts the device name written to the **stdout** file by the Define method to learn the name of the device. If user-specified attributes are supplied with the **-a** flag, the **mkdev** command then invokes the Change method for the device.<br><br>If defining and configuring a device, the **mkdev** command invokes the Define method, gets the name written to the **stdout** file with the Define method, invokes the Change method for the device if user-specified attributes were supplied, and finally invokes the device's Configure method.<br><br>If only configuring a device, the device must already exist in the CuDv object class and its name must be specified to the **mkdev** command. In this case, the **mkdev** command simply invokes the Configure method for the device. |
| **chdev** | The **chdev** command is used to change the characteristics, or attributes, of a device. The device must already exist in the CuDv object class, and the name of the device must be supplied to the **chdev** command. The **chdev** command simply invokes the Change method for the device. |
| **rmdev** | The **rmdev** command can be used to undefine or unconfigure, or unconfigure and undefine, a device. In all cases, the device must already exist in the CuDv object class and the name of the device must be supplied to the **rmdev** command. The **rmdev** command then invokes the Undefine method, the Unconfigure method, or the Unconfigure method followed by the Undefine method, depending on the function requested by the user. |
| **cfgmgr** | The **cfgmgr** command can be used to configure all detectable devices that did not get configured at boot time. This might occur if the devices had been powered off at boot time. The **cfgmgr** command is the Configuration Manager and operates in the same way at run time as it does at boot time. The boot time operation is described in Device Configuration Manager Overview . |

## Understanding Device States

Device methods are responsible for changing the state of a device in the system.

A device can be in one of four states as represented by the Device Status Flag descriptor in the device's object in the Customized Devices (CuDv) object class.

| Item | Description |
|---|---|
| **Defined** | Represented in the Customized database, but neither configured nor available for use in the system. |
| **Available** | Configured and available for use. |
| **Undefined** | Not represented in the Customized database. |

| Item | Description |
| --- | --- |
| **Stopped** | Configured, but not available for use by applications. (Optional state)<br>**Note:** Start and stop methods are only supported on the **inet0** device. |

The Define method is responsible for creating a device instance in the Customized database and setting the state to Defined. The Configure method performs all operations necessary to make the device usable and then sets the state to Available.

The Change method usually does not change the state of the device. If the device is in the Defined state, the Change method applies all changes to the database and leaves the device defined. If the device is in the Available state, the Change method attempts to apply the changes to both the database and the actual device, while leaving the device available. However, if an error occurs when applying the changes to the actual device, the Change method might need to unconfigure the device, thus changing the state to Defined.

Any Unconfigure method you write must perform the operations necessary to make a device unusable. Basically, this method undoes the operations performed by the Configure method and sets the device state to Defined. Finally, the Undefine method actually deletes all information for a device instance from the Customized database, thus reverting the instance to the Undefined state.

The Stopped state is an optional state that some devices require. A device that supports this state needs Start and Stop methods. The Stop method changes the state from Available to Stopped. The Start method changes it from Stopped back to Available.

**Note:** Start and stop methods are only supported on the **inet0** device.

# Adding an Unsupported Device to the System

The operating system provides support for a wide variety of devices. However, some devices are not currently supported.

You can add a currently unsupported device only if you also add the necessary software to support it.

## Modifying the Predefined Database

To add a currently unsupported device to your system, you must modify the Predefined database.

To do this, you must add information about your device to three predefined object classes:

- Predefined Devices (PdDv) object class
- Predefined Attribute (PdAt) object class
- Predefined Connection (PdCn) object class

To describe the device, you must add one object to the PdDv object class to indicate the class, subclass, and device type. You must also add one object to the PdAt object class for each device attribute, such as interrupt level or block size. Finally, you must add objects to the PdCn object class if the device is an intermediate device. If the device is an intermediate device, you must add an object for each different connection location on the intermediate device.

You can use the **odmadd** Object Data Manager (ODM) command from the command line or in a shell script to populate the necessary Predefined object classes from stanza files.

The Predefined database is populated with devices that are present at the time of installation. For some supported devices, such as serial and parallel printers and SCSI disks, the database also contains generic device objects. These generic device objects can be used to configure other similar devices that are not explicitly supported in the Predefined database. If new devices are added after installation, additional device support might need to be installed.

For example, if you have a serial printer that closely resembles a printer supported by the system, and the system's device driver for serial printers works on your printer, you can add the device driver as a printer

of type **osp** (other serial printer). If these generic devices successfully add your device, you do not need to provide additional system software.

### Adding Device Methods

Device methods must be added when adding system support for a new device.

Primary methods needed to support a device are:

- Define
- Configure
- Change
- Undefine
- Unconfigure

When adding a device that closely resembles devices already supported, you might be able to use one of the methods of the already supported device. For example, if you are adding a new type of SCSI disk whose interfaces are identical to supported SCSI disks, the existing methods for SCSI disks may work. If so, all you need to do is populate the Predefined database with information describing the new SCSI disk, which will be similar to information describing a supported SCSI disk.

If you need instructions on how to write a device method, see Writing a Device Method .

### Adding a Device Driver

If you add a new device, you will probably need to add a device driver.

However, if you are adding a new device that closely resembles an already supported device, you might be able to use the existing device driver. For example, when you are adding a new type of SCSI disk whose interfaces are identical to supported SCSI disks, the existing SCSI disk device driver might work.

### Using installp Procedures

The **installp** procedures provide a method for adding the software and Predefined information needed to support your new device.

You might need to write shell scripts to perform tasks such as populating the Predefined database.

## Understanding Device Dependencies and Child Devices

The dependencies that one device has on another can be represented in the configuration database in two ways.

One way usually represents physical connections such as a keyboard device connected to a particular keyboard adapter. The keyboard device has a dependency on the keyboard adapter in that it cannot be configured until after the adapter is configured. This relationship is usually referred to as a parent-child relationship, with the adapter as parent and the keyboard device as child. These relationships are represented with the Parent Device Logical Name and Location Where Device Is Connected descriptors in the Customized Devices (CuDv) object class.

The second method represents a logical connection. A device method can add an object identifying both a dependent device and the device upon which it depends to the Customized Dependency (CuDep) object class. The dependent device is considered to *have* a dependency, and the depended-upon device is considered to *be* a dependency. CuDep objects are usually added to the database to represent a situation in which one device requires access to another device. For example, the lft0 device depends upon a particular keyboard or display device.

These two types of dependencies differ significantly. The configuration process uses parent-child dependencies at boot time to configure all devices that make up a node. The CuDep dependency is usually only used by a device's Configure method to record the names of the devices on which it depends.

For device methods, the parent-child relationship is the more important. Parent-child relationships affect device-method activities in these ways:

- A parent device cannot be unconfigured if it has a configured child.
- A parent device cannot be undefined if it has a defined or configured child.
- A child device cannot be defined if the parent is not defined or configured.
- A child device cannot be configured if the parent is not configured.
- A parent device's configuration cannot be changed if it has a configured child. This guarantees that the information about the parent that the child's device driver might be using remains valid.

However, when a device is listed as a dependency of another device in the CuDep object class, the only effect is to prevent the depended-upon device from being undefined. The name of the dependency is important to the dependent device. If the depended-upon device were allowed to be undefined, a third device could be defined and assigned the same name.

Writers of Unconfigure and Change methods for a depended-upon device should not worry about whether the device is listed as a dependency. If the depended-upon device is actually open by the other device, the Unconfigure and Change operations will fail because their device is busy. But if the depended-upon device is *not* currently open, the Unconfigure or Change operations can be performed without affecting the dependent device.

The possible parent-child connections are defined in the Predefined Connection (PdCn) object class. Each predefined device type that can be a parent device is represented in this object class. There is an object for each connection location (such as slots or ports) describing the subclass of devices that can be connected at that location. The subclass is used to identify each device because it indicates the devices' connection type (for example, SCSI or rs232).

There is no corresponding predefined object class describing the possible CuDep dependencies. A device method can be written so that it already knows what the dependencies are. If predefined data is required, it can be added as predefined attributes for the dependent device in the Predefined Attribute (PdAt) object class.

## Accessing Device Attributes

The predefined device attributes for each type of predefined device are stored in the Predefined Attribute (PdAt) object class.

The objects in the Predefined Attribute (PdAt) object class identify the default values as well as other possible values for each attribute. The Customized Attribute (CuAt) object class contains only attributes for customized device instances that have been changed from their default values.

When a customized device instance is created by a Define method, its attributes assume the default values. As a result, no objects are added to the CuAt object class for the device. If an attribute for the device is changed from the default value by the Change method, an object to describe the attribute's current value is added to the CuAt object class for the attribute. If the attribute is subsequently changed back to the default value, the Change method deletes the CuAt object for the attribute.

Any device methods that need the current attribute values for a device must access both the PdAt and CuAt object classes. If an attribute appears in the CuAt object class, then the associated object identifies the current value. Otherwise, the default value from the PdAt attribute object identifies the current value.

### Modifying an Attribute Value

When modifying an attribute value, methods you write must obtain the objects for that attribute from both the PdAt and CuAt object classes.

Any method you write must be able to handle the following four scenarios:

- If the new value differs from the default value and no object currently exists in the CuAt object class, any method you write must add an object into the CuAt object class to identify the new value.

- If the new value differs from the default value and an object already exists in the CuAt object class, any method you write must update the CuAt object with the new value.
- If the new value is the same as the default value and an object exists in the CuAt object class, any method you write must delete the CuAt object for the attribute.
- If the new value is the same as the default value and no object exists in the CuAt object class, any method you write does not need to do anything.

Your methods can use the **getattr** and **putattr** subroutines to get and modify attributes. The **getattr** subroutine checks both the PdAt and CuAt object classes before returning an attribute to you. It always returns the information in the form of a CuAt object even if returning the default value from the PdAt object class.

Use the **putattr** subroutine to modify these attributes.

# Device Dependent Structure (DDS) Overview

A *device dependent structure* (DDS) contains information that describes a device instance to the device driver. It typically contains information about device-dependent attributes as well as other information the driver needs to communicate with the device.

In many cases, information about a device's parent is included. (For instance, a driver needs information about the adapter and the bus the adapter is plugged into to communicate with a device connected to an adapter.)

A device's DDS is built each time the device is configured. The Configure method can fill in the DDS with fixed values, computed values, and information from the Configuration database. Most of the information from the Configuration database usually comes from the attributes for the device in the Customized Attribute (CuAt) object class, but can come from any of the object classes. Information from the database for the device's parent device or parent's parent device can also be included. The DDS is passed to the device driver with the **SYS_CFGDD** flag of the **sysconfig** subroutine, which calls the device driver's **ddconfig** subroutine with the **CFG_INIT** command.

## How the Change method updates the DDS

The Change method must ensure consistency between the Configuration database and the view that any device driver might have of the device.

The Change method is invoked when changing the configuration of a device. This is accomplished by:

1. Not allowing the configuration to be changed if the device has configured children; that is, children in either the Available or Stopped states. This ensures that a DDS built using information in the database about a parent device remains valid because the parent cannot be changed.
2. If a device has a device driver and the device is in either the Available or Stopped state, the Change method must communicate to the device driver any changes that would affect the DDS. This can be accomplished with **ioctl** operations, if the device driver provides the support to do so. It can also be accomplished by taking the following steps:
   a) Terminating the device instance by calling the **sysconfig** subroutine with the **SYS_CFGDD** operation. This operation calls the device driver's **ddconfig** subroutine with the **CFG_TERM** command.
   b) Rebuilding the DDS using the changed information.
   c) Passing the new DDS to the device driver by calling the **sysconfig** subroutine **SYS_CFGDD** operation. This operation then calls the **ddconfig** subroutine with the **CFG_INIT** command.

Many Change methods simply invoke the device's Unconfigure method, apply changes to the database, and then invoke the device's Configure method. This process ensures the two stipulated conditions since the Unconfigure method, and thus the change, will fail, if the device has Available or Stopped children. Also, if the device has a device driver, its Unconfigure method terminates the device instance. Its Configure method also rebuilds the DDS and passes it to the driver.

## Guidelines for DDS Structure

There is no single defined DDS format. Writers of device drivers and device methods must agree upon a particular device's DDS format. When obtaining information about a parent device, you might want to group that information together in the DDS.

When building a DDS for a device connected to an adapter card, you will typically need the following adapter information:

| Item | Description |
|---|---|
| **slot number** | Obtained from the **connwhere** descriptor of the adapter's Customized Device (CuDv) object. |
| **bus resources** | Obtained from attributes for the adapter in the Customized Attribute (CuAt) or Predefined Attribute (PdAt) object classes. These include attributes for bus interrupt levels, interrupt priorities, bus memory addresses, bus I/O addresses, and DMA arbitration levels. |

The following attribute must be obtained for the adapter's parent bus device:

| Item | Description |
|---|---|
| **bus_id** | Identifies the I/O bus. This field is needed by the device driver to access the I/O bus. |

**Note:** The **getattr** device configuration subroutine should be used whenever attributes are obtained from the Configuration database. This subroutine returns the Customized attribute value if the attribute is represented in the Customized Attribute object class. Otherwise, it returns the default value from the Predefined Attribute object class.

Finally, a DDS generally includes the device's logical name. This is used by the device driver to identify the device when logging an error for the device.

## Example of DDS

The following example provides a guide for using DDS format.

```
/* Device DDS */
struct device_dds {
     /* Bus information */
     ulong  bus_id;         /* I/O bus id                      */
     ushort us_type;        /* Bus type, i.e. BUS_MICRO_CHANNEL*/
     /* Adapter information */
     int   slot_num;         /* Slot number                    */
     ulong io_addr_base;     /* Base bus i/o address           */
     int   bus_intr_lvl;     /* bus interrupt level            */
     int   intr_priority;    /* System interrupt priority      */
     int   dma_lvl;          /* DMA arbitration level          */
     /* Device specific information */
     int   block_size;        /* Size of block in bytes        */
     int   abc_attr;          /* The abc attribute             */
     int   xyz_attr;          /* The xyz attribute             */
     char  resource_name[16]; /* Device logical name           */
};
```

# List of Device Configuration Commands

This section lists the high-level device configuration commands with a brief description.

| Item | Description |
|---|---|
| **chdev** | Changes a device's characteristics. |
| **lsdev** | Displays devices in the system and their characteristics. |
| **mkdev** | Adds a device to the system. |

| Item | Description |
|---|---|
| **rmdev** | Removes a device from the system. |
| **lsattr** | Displays attribute characteristics and possible values of attributes for devices in the system. |
| **lsconn** | Displays the connections a given device, or kind of device, can accept. |
| **lsparent** | Displays the possible parent devices that accept a specified connection type or device. |
| **cfgmgr** | Configures devices by running the programs specified in the Configuration Rules (Config_Rules) object class. |

Associated commands are:

| Item | Description |
|---|---|
| **bootlist** | Alters the list of boot devices seen by ROS when the machine boots. |
| **lscfg** | Displays diagnostic information about a device. |
| **restbase** | Reads the base customized information from the boot image and restores it into the Device Configuration database used during system boot phase 1. |
| **savebase** | Saves information about base customized devices in the Device Configuration Database onto the boot device. |

## List of Device Configuration Subroutines

This section lists the preexisting conditions for using the device configuration library subroutines.

- The caller has initialized the Object Data Manager (ODM) before invoking any of these library subroutines. This is done using the **initialize_odm** subroutine. Similarly, the caller must terminate the ODM (using the **terminate_odm** subroutine) after these library subroutines have completed.
- Because all of these library subroutines (except the **attrval**, **getattr**, and **putattr** subroutines) access the Customized Device Driver (CuDvDr) object class, this class must be exclusively locked and unlocked at the proper times. The application does this by using the **odm_lock** and **odm_unlock** subroutines. In addition, those library subroutines that access the CuDvDr object class exclusively lock this class with their own internal locks.

Following are the device configuration library subroutines:

| Item | Description |
|---|---|
| **attrval** | Verifies that attributes are within range. |
| **genmajor** | Generates the next available major number for a device driver instance. |
| **genminor** | Generates the smallest unused minor number, a requested minor number for a device if it is available, or a set of unused minor numbers. |
| **genseq** | Generates a unique sequence number for creating a device's logical name. |
| **getattr** | Returns attribute objects from either the Predefined Attribute (PdAt) or Customized Attribute (CuAt) object class, or both. |
| **getminor** | Gets from the CuDvDr object class the minor numbers for a given major number. |
| **loadext** | Loads or unloads and binds or unbinds device drivers to or from the kernel. |
| **putattr** | Updates attribute information in the CuAt object class or creates a new object for the attribute information. |
| **reldevno** | Releases the minor number or major number, or both, for a device instance. |
| **relmajor** | Releases the major number associated with a specific device driver instance. |

# Communications I/O Subsystem

The Communication I/O Subsystem consists of one or more physical device handlers (PDHs) that control various communication adapters. The interface to the physical device handlers can support any number of processes, the limit being device-dependent.

The Communication I/O Subsystem design introduces a more efficient, streamlined approach to attaching data link control (DLC) processes to communication and LAN adapters.

**Note:** A PDH, as used for the Communications I/O, provides both the device head role for interfacing to users, and the device handler role for performing I/O to the device.

A communications PDH is a special type of multiplexed character device driver. Information common to all communications device handlers is discussed here. Additionally, individual communications PDHs have their own adapter-specific sets of information. Refer to the following to learn more about the adapter types:

- MPQP Device Handler Interface Overview
- Serial Optical Link Device Handler Overview

Each adapter type requires a device driver. Each PDH can support one or more adapters of the same type.

There are two interfaces a user can use to access a PDH. One is from a user-mode process (application space), and the other is from a kernel-mode process (within the kernel).

**Related concepts**

Common Communications Status and Exception Codes
In general, communication device handlers return codes from a group of common exception codes. However, device handlers for specific communication devices can return device-specific exception codes.

Logical File System Kernel Services
The Logical File System services (also known as the **fp_**services) allow processes running in kernel mode to open and manipulate files in the same way that user-mode processes do. Data access limitations make it unreasonable to accomplish these tasks with system calls, so a subset of the file system calls has been provided with an alternate kernel-only interface.

**Related information**

System Management Interface Tool (SMIT)
Error Logging Overview

## User-Mode Interface to a Communications PDH

The user-mode process uses system calls (**open**, **close**, **select**, **poll**, **ioctl**, **read**, **write**) to interface to the PDH to send or receive data.

The **poll** or **select** subroutine notifies a user-mode process of available receive data, available transmit, and status and exception conditions.

## Kernel-Mode Interface to a Communications PDH

The kernel-mode interface to a communications PDH differs from the interface supported for a user-mode process in the following ways:

- Kernel services are used instead of system calls. This means that, for example, the **fp_open** kernel service is used instead of the **open** subroutine. The same holds true for the **fp_close**, **fp_ioctl**, and **fp_write** kernel services.
- The **ddread** entry point, **ddselect** entry point, and **CIO_GET_STAT** (Get Status) ddioctl operation are not supported in kernel mode. Instead, kernel-mode processes specify at open time the addresses of their own procedures for handling receive data available, transmit available and status or exception conditions. The PDH directly calls the appropriate procedure, whenever that condition arises. These kernel procedures must execute and return quickly since they are executing within the priority of the PDH.

- The **ddwrite** operation for a kernel-mode process differs from a user-mode process in that there are two ways to issue a **ddwrite** operation to transmit data:
  - Transmit each buffer of data with the **fp_write** kernel service.
  - Use the fast write operation, which allows the user to directly call the **ddwrite** operation (no context switching) for each buffer of data to be transmitted. This operation helps increase the performance of transmitted data. A **fp_ioctl** (**CIO_GET_FASTWRT**) kernel service call obtains the functional address of the write function. This address is used on all subsequent write function calls. Support of the fast write operation is optional for each device.

## CDLI Device Drivers

Some device drivers have a different design and use the services known as Common Data Link Interface (CDLI).

The following device drivers use CDLI:

- Forum-Compliant ATM LAN Emulation Device Driver
- Ethernet Device Drivers

## Communications Physical Device Handler Model Overview

A physical device handler (PDH) must provide eight common entry points. An individual PDH names its entry points by placing a unique identifier in front of the supported command type.

The following are the required eight communications PDH entry points:

| Item | Description |
|---|---|
| **ddconfig** | Performs configuration functions for a device handler. Supported the same way that the common **ddconfig** entry point is. |
| **ddmpx** | Allocates or deallocates a channel for a multiplexed device handler. Supported the same way as the common **ddmpx** device handler entry point. |
| **ddopen** | Performs data structure allocation and initialization for a communications PDH. Supported the same way as the common **ddopen** entry point. Time-consuming tasks, such as port initialization and connection establishment, are deferred until the (**CIO_START**) ddioctl call is issued. A PDH can support multiple users of a single port. |
| **ddclose** | Frees up system resources used by the specified communications device until they are needed again. Supported the same way as the common **ddclose** entry point. |
| **ddwrite** | Queues a message for transmission or blocks until the message can be queued. The **ddwrite** entry point can attempt to queue a transmit request (nonblocking) or wait for it to be queued (blocking), depending on the setting of the **DNDELAY** flag. The caller has the additional option of requesting an asynchronous acknowledgment when the transmission actually completes. |
| **ddread** | Returns a message of data to a user-mode process. Supports blocking or nonblocking reads depending on the setting of the **DNDELAY** flag. A blocking read request does not return to the caller until data is available. A nonblocking read returns with a message of data if it is immediately available. Otherwise, it returns a length of 0 (zero). |
| **ddselect** | Checks to see if a specified event or events has occurred on the device for a user-mode process. Supported the same way as the common **ddselect** entry point. |

| Item | Description |
|---|---|
| **ddioctl** | Performs the special I/O operations requested in an **ioctl** subroutine. Supported the same way as the common **ddioctl** entry point. In addition, a communications PDH must support the following four options:<br>• **CIO_START**<br>• **CIO_HALT**<br>• **CIO_QUERY**<br>• **CIO_GET_STAT** |

Individual PDHs can add additional commands. Hardware initialization and other time-consuming activities, such as call establishment, are performed during the **CIO_START** operation.

## Use of mbuf Structures in the Communications PDH

PDHs use **mbuf** structures to buffer send and receive data. These structures allow the PDH to gather data when transmitting frames and scatter for receive operations. The **mbuf** structures are internal to the kernel and are used only by kernel-mode processes and PDHs.

PDHs and kernel-mode processes require a set of utilities for obtaining and returning **mbuf** structures from a buffer pool.

Kernel-mode processes use the Berkeley **mbuf** scheme for transmit and receive buffers. The structure for an **mbuf** is defined in the **/usr/include/sys/mbuf.h** file.

## Common Communications Status and Exception Codes

In general, communication device handlers return codes from a group of common exception codes. However, device handlers for specific communication devices can return device-specific exception codes.

Common exception codes are defined in the **/usr/include/sys/comio.h** file and include the following:

| Item | Description |
|---|---|
| **CIO_OK** | Indicates that the operation was successful. |
| **CIO_BUF_OVFLW** | Indicates that the data was lost due to buffer overflow. |
| **CIO_HARD_FAIL** | Indicates that a hardware failure was detected. |
| **CIO_NOMBUF** | Indicates that the operation was unable to allocate **mbuf** structures. |
| **CIO_TIMEOUT** | Indicates that a time-out error occurred. |
| **CIO_TX_FULL** | Indicates that the transmit queue is full. |
| **CIO_NET_RCVRY_ENTER** | Enters network recovery. |
| **CIO_NET_RCVRY_EXIT** | Indicates the device handler is exiting network recovery. |
| **CIO_NET_RCVRY_MODE** | Indicates the device handler is in Recovery mode. |
| **CIO_INV_CMD** | Indicates that an invalid command was issued. |
| **CIO_BAD_MICROCODE** | Indicates that the microcode download failed. |
| **CIO_NOT_DIAG_MODE** | Indicates that the command could not be accepted because the adapter is not open in Diagnostic mode. |
| **CIO_BAD_RANGE** | Indicates that the parameter values have failed a range check. |
| **CIO_NOT_STARTED** | Indicates that the command could not be accepted because the device has not yet been started by the first call to **CIO_START** operation. |
| **CIO_LOST_DATA** | Indicates that the receive packet was lost. |

| Item | Description |
|---|---|
| **CIO_LOST_STATUS** | Indicates that a status block was lost. |
| **CIO_NETID_INV** | Indicates that the network ID was not valid. |
| **CIO_NETID_DUP** | Indicates that the network ID was a duplicate of an existing ID already in use on the network. |
| **CIO_NETID_FULL** | Indicates that the network ID table is full. |

**Related concepts**

Communications I/O Subsystem

The Communication I/O Subsystem consists of one or more physical device handlers (PDHs) that control various communication adapters. The interface to the physical device handlers can support any number of processes, the limit being device-dependent.

## Status Blocks for Communications Device Handlers Overview

Status blocks are used to communicate status and exception information.

User-mode processes receive a status block whenever they request a **CIO_GET_STAT** operation. A user-mode process can wait for the next available status block by issuing a **ddselect** entry point with the specified **POLLPRI** event.

A kernel-mode process receives a status block through the **stat_fn** procedure. This procedure is specified when the device is opened with the **ddopen** entry point.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, **CIO_START_DONE**). A status block's options depend on the block code. The C structure of a status block is defined in the **/usr/include/sys/comio.h** file.

Additional device-dependent status block codes may be defined.

### CIO_START_DONE

The **CIO_START** operation provides the following block on completion.

This block is provided by the device handler when the **CIO_START** operation completes:

| Item | Description |
|---|---|
| `option[0]` | The **CIO_OK** or **CIO_HARD_FAIL** status/exception code from the common or device-dependent list. |
| `option[1]` | The low-order two bytes are filled in with the `netid` field. This field is passed when the **CIO_START** operation is invoked. |
| `option[2]` | Device-dependent. |
| `option[3]` | Device-dependent. |

### CIO_HALT_DONE

The **CIO_HALT** operation provides the following block on completion.

This block is provided by the device handler when the **CIO_HALT** operation completes:

| Item | Description |
|---|---|
| `option[0]` | The **CIO_OK** status/exception code from the common or device-dependent list. |
| `option[1]` | The low-order two bytes are filled in with the `netid` field. This field is passed when the **CIO_START** operation is invoked. |
| `option[2]` | Device-dependent. |

| Item | Description |
|---|---|
| option[3] | Device-dependent. |

## CIO_TX_DONE

The block that is listed in this section is provided when the physical device handler (PDH) is finished with a transmit request for which acknowledgment was requested.

| Item | Description |
|---|---|
| option[0] | The **CIO_OK** or **CIO_TIMEOUT** status/exception code from the common or device-dependent list. |
| option[1] | The write_id field specified in the **write_extension** structure passed in the *ext* parameter to the **ddwrite** entry point. |
| option[2] | For a kernel-mode process, indicates the **mbuf** pointer for the transmitted frame. |
| option[3] | Device-dependent. |

## CIO_NULL_BLK

This block is returned whenever a status block is requested but there are none available:

| Item | Description |
|---|---|
| option[0] | Not used |
| option[1] | Not used |
| option[2] | Not used |
| option[3] | Not used |

## CIO_LOST_STATUS

This block is returned once after one or more status blocks is lost due to status queue overflow.

The **CIO_LOST_STATUS** block provides the following:

| Item | Description |
|---|---|
| option[0] | Not used |
| option[1] | Not used |
| option[2] | Not used |
| option[3] | Not used |

## CIO_ASYNC_STATUS

This status block is used to return status and exception codes that occur unexpectedly:

| Item | Description |
|---|---|
| option[0] | The **CIO_HARD_FAIL** or **CIO_LOST_DATA** status/exception code from the common or device-dependent list |
| option[1] | Device-dependent |
| option[2] | Device-dependent |
| option[3] | Device-dependent |

# MPQP Device Handler Interface Overview for the ARTIC960Hx PCI Adapter

The PCI MPQP device handler interface is made up of the following eight entry points:

The ARTIC960Hx PCI Adapter (PCI MPQP) device handler is a component of the communication I/O subsystem.

| Item | Description |
|---|---|
| **tsclose** | Resets the PCI MPQP device to a known state and returns system resources back to the system on the last close for that adapter. The port no longer transmits or receives data. |
| **tsconfig** | Provides functions for initializing and terminating the PCI MPQP device handler and adapter. |
| **tsopen** | Opens a channel on the PCI MPQP device for transmitting and receiving data. |
| **tsmpx** | Provides allocation and deallocation of a channel. |
| **tsread** | Provides the means for receiving data to the PCI MPQP device. |
| **tsselect** | Provides the means for determining which specified events have occurred on the PCI MPQP device. |
| **tswrite** | Provides the means for transmitting data to the PCI MPQP device. |

## Binary Synchronous Communication (BSC) with the PCI MPQP Adapter

The PCI MPQP adapter software performs low-level BSC frame-type determination to facilitate character parsing at the kernel-mode process level.

Frames received without errors are parsed. A message type is returned in the `status` field of the extension block along with a pointer to the receive buffer. The message type indicates the type of frame that was received.

For control frames that only contain control characters, the message type is returned and no data is transferred from the board. For example, if an ACK0 was received, the message type `MP_ACK0` is returned in the `status` field of the extension block. In addition, a NULL pointer for the receive buffer is returned. If an error occurs, the error status is logged by the device driver. Unlogged buffer overrun errors are an exception.

**Note:** In BSC communications, the caller receives either a message type or an error status.

Read operations must be performed using the **readx** subroutine because the **read_extension** structure is needed to return BSC function results.

### *BSC Message Types Detected by the PCI MPQP Adapter*

BSC message types are defined in the **/usr/include/sys/mpqp.h** file.

The PCI MPQP adapter can detect the following message types:

| Item | Description | |
|---|---|---|
| **MP_ACK0** | **MP_DISC** | **MP_STX_ETX** |
| **MP_ACK1** | **MP_SOH_ITB** | **MP_STX_ENQ** |
| **MP_WACK** | **MP_SOH_ETB** | **MP_DATA_ACK0** |
| **MP_NAK** | **MP_SOH_ETX** | **MP_DATA_ACK1** |
| **MP_ENQ** | **MP_SOH_ENQ** | **MP_DATA_NAK** |
| **MP_EOT** | **MP_STX_ITB** | **MP_DATA_ENQ** |
| **MP_RVI** | **MP_STX_ETB** | |

### *Receive Errors Logged by the PCI MPQP Adapter*

The PCI MPQP adapter detects many types of receive errors. As errors occur they are logged and the appropriate statistical counter is incremented. The kernel-mode process is not notified of the error.

The following are the possible BSC receive errors logged by the PCI MPQP adapter:

- Receive overrun
- A cyclical redundancy check (CRC) or longitudinal redundancy check (LRC) framing error
- Parity error
- Clear to Send (CTS) timeout
- Data synchronization lost
- ID field greater than 15 bytes (BSC)
- Invalid pad at end of frame (BSC)
- Unexpected or invalid data (BSC)

If status and data information are available, but no extension block is provided, the **read** operation returns the data, but not the status information.

**Note:** Errors, such as buffer overflow errors, can occur during the read data operation. In these cases, the return value is the byte count. Therefore, status should be checked even if no **errno** global value is returned.

## Description of the PCI MPQP Card

The PCI MPQP card is a 4-port multiprotocol adapter that supports BSC and SDLC on the EIA232-D, X.21, and V.35 physical interfaces.

When using the X.21 physical interface, X.21 centralized multipoint operation on a leased-circuit public data network is not supported.

# Serial Optical Link Device Handler Overview

The serial optical link (SOL) device handler is a component of the communication I/O subsystem.

The device handler can support one to four serial optical ports. An optical port consists of two separate pieces. The serial link adapter is on the system planar and is packaged with two to four adapters in a single chip. The serial optical channel converter plugs into a slot on the system planar and provides two separate optical ports.

## Special Files

There are two separate interfaces to the serial optical link device handler. The special file **/dev/ops0** provides access to the optical port subsystem. An application that opens this special file has access to all the ports, but it does not need to be aware of the number of ports available. Each write operation includes a destination processor ID. The device handler sends the data out the correct port to reach that processor. In case of a link failure, the device handler uses any link that is available.

The **/dev/op0**, **/dev/op1**, ..., **/dev/op***n* special files provide a diagnostic interface to the serial link adapters and the serial optical channel converters. Each special file corresponds to a single optical port that can only be opened in Diagnostic mode. A diagnostic open allows the diagnostic ioctls to be used, but normal reads and writes are not allowed. A port that is open in this manner cannot be opened with the **/dev/ops0** special file. In addition, if the port has already been opened with the **/dev/ops0** special file, attempting to open a **/dev/op***x* special file will fail unless a forced diagnostic open is used.

# Configuring the Serial Optical Link Device Driver

When configuring the serial optical link (SOL) device driver, consider the physical and logical devices.

Also, consider the changeable attributes of the SOL subsystem.

## Physical and Logical Devices

The SOL subsystem consists of several physical and logical devices in the ODM configuration database.

| Device | Description |
|---|---|
| **slc** (serial link chip) | There are two serial link adapters in each COMBO chip. The **slc** device is automatically detected and configured by the system. |
| **otp** (optic two-port card) | Also known as the serial optical channel converter (SOCC). There is one SOCC possible for each **slc**. The **otp** device is automatically detected and configured by the system. |
| **op** (optic port) | There are two optic ports per **otp**. The **op** device is automatically detected and configured by the system. |
| **ops** (optic port subsystem) | This is a logical device. There is only one created at any time. The **ops** device requires some additional configuration initially, and is then automatically configured from that point on. The **/dev/ops0** special file is created when the **ops** device is configured. The **ops** device cannot be configured when the processor ID is set to -1. |

## Changeable Attributes of the Serial Optical Link Subsystem

The system administrator can change the attributes of the serial optical link subsystem that are explained in this section.

**Note:** If your system uses serial optical link to make a direct, point-to-point connection to another system or systems, special conditions apply. You must start interfaces on two systems at approximately the same time, or a method error occurs. If you wish to connect to at least one machine on which the interface has already been started, this is not necessary.

| Item | Description |
|---|---|
| **Processor ID** | This is the address by which other machines connected by means of the optical link address this machine. The processor ID can be any value in the range of 1 to 254. To avoid a conflict on the network, this value is initially set to -1, which is not valid, and the **ops** device cannot be configured.<br><br>**Note:** If you are using TCP/IP over the serial optical link, the processor ID must be the same as the low-order octet of the IP address. It is not possible to successfully configure TCP/IP if the processor ID does not match. |
| **Receive Queue Size** | This is the maximum number of packets that is queued for a user-mode caller. The default value is 30 packets. Any integer in the range from 30 to 150 is valid. |
| **Status Queue Size** | This is the maximum number of status blocks that will be queued for a user-mode caller. The default value is 10. Any integer in the range from 3 to 20 is valid. |

The standard SMIT interface is available for setting these attributes, listing the serial optical channel converters, handling the initial configuration of the **ops** device, generating a trace report, generating an error report, and configuring TCP/IP.

## Forum-Compliant ATM LAN Emulation Device Driver

The **Forum-Compliant ATM LAN Emulation** (**LANE**) device driver allows communications applications and access methods that would normally operate over local area network (LAN) attachments to operate over high-speed ATM networks.

This **ATM LANE** function supports LAN Emulation Client (LEC) as specified in *The ATM Forum Technical Committee LAN Emulation Over ATM Version 1.0*, as well as MPOA Client (MPC) through a subset of *ATM Forum LAN Emulation Over ATM Version 2 - LUNI Specification*, and *ATM Forum Multi-Protocol Over ATM Version 1.0*.

The **ATM LANE** device driver emulates the operation of Standard Ethernet, IEEE 802.3 Ethernet, and IEEE 802.5 Token Ring LANs. It encapsulates each LAN packet and transfers its LAN data over an ATM network at up to OC12 speeds (622 megabits per second). This data can also be bridged transparently to a traditional LAN with ATM/LAN bridges such as the IBM® 2216.

Each LEC participates in an emulated LAN containing additional functions such as:

- A LAN Emulation Configuration Server (LECS) that provides automated configuration of the LEC's operational attributes.
- A LAN Emulation Server (LES) that provides address resolution
- A Broadcast and Unknown Server (BUS) that distributes packets sent to a broadcast address or packets sent without knowing the ATM address of the remote station (for example, whenever an ARP response has not been received yet).

There is always at least one ATM switch and a possibility of additional switches, bridges, or concentrators.

ATM supports UNI3.0, UNI3.1, and UNI4.0 signalling.

In support of Ethernet jumbo frames, LE Clients can be configured with maximum frame size values greater than 1516 bytes. Supported forum values are: 1516, 4544, 9234, and 18190.

Incoming Add Party requests are supported for the Control Distribute and Multicast Forward Virtual Circuits (VCs). This allows multiple LE clients to be open concurrently on the same ELAN without additional hardware.

LANE and MPOA are both enabled for IPV4 TCP checksum offload. Transmit offload is automatically enabled when it is supported by the adapter. Receive offload is configured by using the rx_checksum attribute. The NDD_CHECKSUM_OFFLOAD device driver flag is set to indicate general offload capability and also indicates that transmit offload is operational.

Transmit offload of IP-fragmented TCP packets is not supported. Transmit packets that MPOA needs to fragment are offloaded in the MPOA software, instead of in the adapter. UDP offloading is also not supported.

The **ATM LANE** device driver is a dynamically loadable device driver. Each LE Client or MPOA Client is configurable by the operator, and the LANE driver is loaded into the system as part of that configuration process. If an LE Client or MPOA Client has already been configured, the LANE driver is automatically reloaded at reboot time as part of the system configuration process.

The interface to the **ATM LANE** device driver is through kernel services known as Network Services.

Interfacing to the **ATM LANE** device driver is achieved by calling the device driver's entry points for opening the device, closing the device, transmitting data, and issuing device control commands, just as you would interface to any of the Common Data Link Interface (CDLI) LAN device drivers.

The **ATM LANE** device driver interfaces with all hardware-level ATM device drivers that support CDLI, ATM Call Management, and ATM Signaling.

## Adding ATM LANE Clients

At least one ATM LAN Emulation client must be added to the system to communicate over an ATM network using the **ATM Forum LANE** protocol. A user with root authority can add Ethernet or Token-Ring clients using the **smit atmle_panel** fast path.

Entries are required for the Local LE Client's **LAN MAC Address** field and possibly the **LES ATM Address** or **LECS ATM Address** fields, depending on the support provided at the server. If the server accepts the well-known ATM address for LECS, the value of the **Automatic Configuration via LECS** field can be set to **Yes**, and the **LES** and **LECS ATM Address** fields can be left blank. If the server does not support the well-known ATM address for LECS, an ATM address must be entered for either LES (manual configuration) or LECS (automatic configuration). All other configuration attribute values are optional. If used, you can accept the defaults for ease-of-use.

Configuration help text is also available within the SMIT LE Client add and change menus.

## Configuration Parameters for the ATM LANE Device Driver

The **ATM LANE** device driver supports the following configuration parameters for each LE Client:

| Item | Description |
|---|---|
| **addl_drvr** | Specifies the CDLI demultiplexer being used by the LE Client. The value set by the **ATM LANE** device driver is **/usr/lib/methods/cfgdmxtok** for Token Ring emulation and **/usr/lib/methods/cfgdmxeth** for Ethernet. This is not an operator-configurable attribute. |
| **addl_stat** | Specifies the routine being used by the LE client to generate device-specific statistics for the **entstat** and **tokstat** commands. The values set by the **ATM LANE** device driver are:<br>• **/usr/sbin/atmle_ent_stat**<br>• **/usr/sbin/atmle_tok_stat**<br>The **addl_stat** attribute is not operator-configurable. |
| **arp_aging_time** | Specifies the maximum timeout period (in seconds) that the LE Client maintains an LE_ARP cache entry without verification (ATM Forum LE Client parameter *C17*). The default value is 300 seconds. |
| **arp_cache_size** | Specifies the maximum number of LE_ARP cache entries that are held by the LE Client before removing the least recently used entry. The default value is 128 entries. |
| **arp_response_timeout** | Specifies the maximum timeout period (in seconds) for LE_ARP request/response exchanges (ATM Forum LE Client parameter *C20*). The default value is 1 second. |
| **atm_device** | Specifies the logical name of the physical **ATM** device driver that this LE Client is to operate with, as specified in the CuDv database (for example, **atm0**, **atm1**, **atm2**, ...). The default is **atm0**. |

| Item | Description |
| --- | --- |
| **auto_cfg** | Specifies whether the LE Client is to be automatically configured. Select **Yes** if the LAN Emulation Configuration Server (LECS) is to be used by the LE Client to obtain the ATM address of the LE ARP Server, as well as any additional configuration parameters provided by the LECS. The default value is No (manual configuration). The attribute values are:<br>**Yes**<br>auto configuration<br>**No**<br>manual configuration<br>**Note:** Configuration parameters provided by LECS override configuration values provided by the operator. |
| **debug_trace** | Specifies whether this LE Client should keep a real time debug log within the kernel and allow full system trace capability. Select **Yes** to enable full tracing capability for this LE Client. Select **No** for optimal performance when minimal tracing is desired. The default is **Yes** (full tracing capability). |
| **elan_name** | Specifies the name of the Emulated LAN this LE Client wishes to join (ATM Forum LE Client parameter *C5*). This is an SNMPv2 DisplayString of 1-32 characters, or it might be left blank (unused). See RFC1213 for a definition of an SNMPv2 DisplayString.<br>**Note:**<br>1. Any operator configured **elan_name** should match exactly what is expected at the LECS/LES server when attempting to join an ELAN. Some servers can alias the ELAN name and allow the operator to specify a logical name that correlates to the actual name. Other servers might require the exact name to be specified.<br>Previous versions of LANE would accept any **elan_name** from the server, even when configured differently by the operator. However, with multiple LECS/LES now possible, it is desirable that only the ELAN identified by the network administrator is joined. Use the **force_elan_name** attribute below to ensure that the name you have specified is the only ELAN joined.<br>If no **elan_name** attribute is configured at the LEC, or the **force_elan_name** attribute is disabled, the server can stipulate whatever **elan_name** is available.<br>Failure to use an ELAN name that is identical to the server's when specifying the **elan_name** and **force_elan_name** attributes causes the LEC to fail the join process, with **entstat**/**tokstat** status indicating Driver Flag **Limbo**.<br>2. Blanks can be inserted within an **elan_name** by typing a tilde (~) character whenever a blank character is desired. This allows a network administrator to specify an ELAN name with imbedded blanks as in the default of some servers.<br>Any tilde (~) character that occupies the first character position of the **elan_name** remains unchanged (that is, the resulting name can start with a tilde (~) but all remaining tilde characters are converted to blanks). |

| Item | Description |
| --- | --- |
| **failsafe_time** | Specifies the maximum timeout period (in seconds) that the LE Client attempts to recover from a network outage. A value of zero indicates that you should continue recovery attempts unless a nonrecoverable error is encountered. The default value is 0 (unlimited). |
| **flush_timeout** | Specifies the maximum timeout period (in seconds) for FLUSH request/response exchanges (ATM Forum LE Client parameter *C21*). The default value is 4 seconds. |
| **force_elan_name** | Specifies that the Emulated LAN Name returned from the LECS or LES servers must exactly match the name entered in the **elan_name** attribute above. Select **Yes** if the **elan_name** field must match the server configuration and join parameters. This allows a specific ELAN to be joined when multiple LECS and LES servers are available on the network. The default value is No, which allows the server to specify the ELAN Name. |
| **fwd_delay_time** | Specifies the maximum timeout period (in seconds) that the LE Client maintains an entry for a non-local MAC address in its LE_ARP cache without verification, when the **Topology Change** flag is True (ATM Forum LE Client parameter *C18*). The default value is 15 seconds. |
| **fwd_dsc_timeout** | Specifies the timeout period (in seconds) that can elapse without an active Multicast Forward VCC from the BUS. (ATM Forum LE Client parameter C33). If the timer expires without an active Multicast Forward VCC, the LE Client attempts recovery by re-establishing its Multicast Send VCC to the BUS. The default value is 60 seconds. |
| **init_ctl_time** | Specifies the initial control timeout period (in seconds) for most request/response control frame interactions (ATM Forum LE Client parameter C7i). This timeout is increased by its initial value after each timeout expiration without a response, but does not exceed the value specified by the Maximum Control Timeout attribute (max_ctl_time). The default value is 5 seconds. |
| **lan_type** | Identifies the type of local area network being emulated (ATM Forum LE Client parameter *C2*). Both Ethernet/IEEE 802.3 and Token Ring LANs can be emulated using ATM Forum LANE. The attribute values are:<br>• Ethernet/IEEE802.3<br>• TokenRing |
| **lecs_atm_addr** | If you are doing auto configuration using the **LE Configuration Server** (**LECS**), this field specifies the ATM address of LECS. It can remain blank if the address of LECS is not known and the LECS is connected by way of PVC (VPI=0, VCI=17) or the well-known address, or is registered by way of ILMI. If the 20-byte address of the LECS is known, it must be entered as hexadecimal numbers using a period (.) as the delimiter between bytes. Leading zeros of each byte can be omitted, for example:<br>`47.0.79.0.0.0.0.0.0.0.0.0.0.0.a0.3.0.0.1`<br>`(the LECS well-known address)` |

| Item | Description |
| --- | --- |
| **les_atm_addr** | If you are doing manual configuration (without the aid of an **LECS**), this field specifies the ATM address of the LE ARP Server (LES) (ATM Forum LE Client parameter *C9*). This 20-byte address must be entered as hexadecimal numbers using a period (.) as the delimiter between bytes. Leading zeros of each byte can be omitted, for example:<br>`39.11.ff.22.99.99.99.0.0.0.0.1.49.10.0.5a.68.0.a.1` |
| **local_lan_addrs** | Specifies the local unicast LAN MAC address that is represented by this LE Client and registered with the LE Server (ATM Forum LE Client parameter *C6*). This 6-byte address must be entered as hexadecimal numbers using a period (.) as the delimiter between bytes. Leading zeros of each byte can be omitted.<br>`Ethernet Example: 2.60.8C.2C.D2.DC`<br>`Token Ring Example: 10.0.5A.4F.4B.C4` |
| **max_arp_retries** | Specifies the maximum number of times an LE_ARP request can be retried (ATM Forum LE Client parameter *C13*). The default value is 1. |
| **max_config_retries** | Specifies the number of times a configuration control frame such as LE_JOIN_REQUEST should be retried. Duration (in seconds) between retries is derived from the **init_ctl_time** and **max_ctl_time** attributes. The default is 1. |
| **max_ctl_time** | Specifies the maximum timeout period (in seconds) for most request and response control frame interactions (ATM Forum LE Client parameter C7). The default value is 30 seconds. |
| **max_frame_size** | Specifies the maximum AAL-5 send data-unit size of data frames for this LE Client. In general, this value should coincide with the LAN type and speed as follows:<br>**Unspecified**<br>for auto LECS configuration<br>**1516 bytes**<br>for Ethernet and IEEE 802.3 networks<br>**4544 bytes**<br>for 4 Mbps Token Rings or Ethernet jumbo frames<br>**9234 bytes**<br>for 16 Mbps Token Rings or Ethernet jumbo frames<br>**18190 bytes**<br>for 16 Mbps Token Rings or Ethernet jumbo frames |
| **max_queued_frames** | Specifies the maximum number of outbound packets that are held for transmission per LE_ARP cache entry. This queueing occurs when the Maximum Unknown Frame Count (**max_unknown_fct**) has been reached, or when flushing previously transmitted packets while switching to a new virtual channel. The default value is 60 packets. |
| **max_rdy_retries** | Specifies the maximum number of READY_QUERY packets sent in response to an incoming call that has not yet received data or a READY_IND packet. The default value is 2 retries. |

| Item | Description |
|---|---|
| **max_unknown_fct** | Specifies the maximum number of frames for a given unicast LAN MAC address that can be sent to the Broadcast and Unknown Server (BUS) within time period Maximum Unknown Frame Time (**max_unknown_ftm**) (ATM Forum LE Client parameter *C10*). The default value is 1. |
| **max_unknown_ftm** | Specifies the maximum timeout period (in seconds) that a given unicast LAN address can be sent to the Broadcast and Unknown Server (BUS). The LE Client sends no more than Maximum Unknown Frame Count (**max_unknown_fct**) packets to a given unicast LAN destination within this timeout period (ATM Forum LE Client parameter *C11*). The default value is 1 second. |
| **mpoa_enabled** | Specifies whether Forum MPOA and LANE-2 functions should be enabled for this LE Client. Select **Yes** if MPOA will be operational on the LE Client. Select **No** when traditional LANE-1 functionality is required. The default is No (LANE-1). |
| **mpoa_primary** | Specifies whether this LE Client is to be the primary configurator for MPOA using LAN Emulation Configuration Server (LECS). Select **Yes** if this LE Client will be obtaining configuration information from the LECS for the MPOA Client. This attribute is only meaningful if running auto config with an LECS, and indicates that the MPOA configuration TLVs from this LEC is available to the MPC. Only one LE Client can be active as the MPOA primary configurator. The default is No. |
| **path_sw_delay** | Specifies the maximum timeout period (in seconds) that frames sent on any path in the network take to be delivered (ATM Forum LE Client parameter *C22*). The default value is 6 seconds. |
| **peak_rate** | Specifies the forward and backward peak bit rate in K-bits per second that are used by this LE Client to set up virtual channels. Specify a value that is compatible with the lowest speed remote device with which you expect this LE Client to be communicating. Higher values might cause congestion in the network. A value of zero allows the LE Client to adjust its peak_rate to the actual speed of the adapter. If the adapter does not provide its maximum peak rate value, the LE Client defaults its peak_rate to 25600. Any non-zero value specified is accepted and used by the LE Client up to the maximum value allowed by the adapter. The default value is 0, which uses the adapter's maximum peak rate. |
| **ready_timeout** | Specifies the maximum timeout period (in seconds) in which data or a READY_IND message is expected from a calling party (ATM Forum LE Client parameter *C28*). The default value is 4 seconds. |
| **ring_speed** | Specifies the Token Ring speed as viewed by the ifnet layer. The value set by the **ATM LANE** device driver is 16 Mbps for Token Ring emulation and ignored for Ethernet. This is not an operator-configurable attribute. |
| **rx_checksum** | Specifies whether this LE Client should offload TCP receive checksums to the ATM hardware. Select **Yes** if TCP checksums should be handled in hardware. Select No if TCP checksums should be handled in software. The default is **Yes** (enable hardware receive checksum).<br><br>**Note:** The ATM adapter must also have receive checksum enabled to be functional. |

| Item | Description |
| --- | --- |
| **soft_restart** | Specifies whether active data virtual circuits (VCs) are to be maintained during connection loss of ELAN services such as the LE ARP Server (LES) or Broadcast and Unknown Server (BUS). Normal ATM Forum operation forces a disconnect of data VCs when LES/BUS connections are lost. This option to maintain active data VCs might be advantageous when server backup capabilities are available. The default value is No. |
| **vcc_activity_timeout** | Specifies the maximum timeout period (in seconds) for inactive Data Direct Virtual Channel Connections (VCCs). Any switched Data Direct VCC that does not transmit or receive data frames in this timeout period is terminated (ATM Forum LE Client parameter *C12*). The default value is 1200 seconds (20 minutes). |

## Device Driver Configuration and Unconfiguration

The **atmle_config** entry point performs configuration functions for the **ATM LANE** device driver.

Configuration help text is also available within the SMIT LE Client add and change menus.

## Device Driver Open

The **atmle_open** function is called to open the specified network device.

The **LANE** device driver does an asynchronous open. It starts the process of attaching the device to the network, sets the **NDD_UP** flag in the **ndd_flags** field, and returns 0. The network attachment continues in the background where it is driven by network activity and system timers.

> **Note:** The Network Services **ns_alloc** routine that calls this open routine causes the open to be synchronous. It waits until the **NDD_RUNNING** or the **NDD_LIMBO** flag is set in the **ndd_flags** field or 15 seconds have passed.

If the connection is successful, the **NDD_RUNNING** flag is set in the **ndd_flags** field, and an NDD_CONNECTED status block is sent. The **ns_alloc** routine returns at this time.

If the device connection fails, the **NDD_LIMBO** flag is set in the **ndd_flags** field, and an NDD_LIMBO_ENTRY status block is sent.

If the device is eventually connected, the **NDD_LIMBO** flag is disabled, and the **NDD_RUNNING** flag is set in the **ndd_flags** field. Both NDD_CONNECTED and NDD_LIMBO_EXIT status blocks are sent.

## Device Driver Close

The **atmle_close** function is called by the Network Services **ns_free** routine to close the specified network device. This function resets the device to a known state and frees system resources associated with the device.

The device will not be detached from the network until the device's transmit queue is allowed to drain.

## Data Transmission

The **atmle_output** function transmits data using the network device.

If the destination address in the packet is a broadcast address, the **M_BCAST** flag in the **p_mbuf->m_flags** field should be set prior to entering this routine. A broadcast address is defined as FF.FF.FF.FF.FF.FF (hex) for both Ethernet and Token Ring and C0.00.FF.FF.FF.FF (hex) for Token Ring.

If the destination address in the packet is a multicast or group address, the **M_MCAST** flag in the **p_mbuf->m_flags** field should be set prior to entering this routine. A multicast or group address is defined as any nonindividual address other than a broadcast address.

The device driver keeps statistics based on the **M_BCAST** and **M_MCAST** flags.

Token Ring LANE emulates a duplex device. If a Token Ring packet is transmitted with a destination address that matches the LAN MAC address of the local LE Client, the packet is received. This is also True for Token Ring packets transmitted to a broadcast address, enabled functional address, or an enabled group address. Ethernet LANE, on the other hand, emulates a simplex device and does not receive its own broadcast or multicast transmit packets.

## Data Reception

When the **LANE** device driver receives a valid packet from a network ATM device driver, the **LANE** device driver calls the **nd_receive** function that is specified in the **ndd_t** structure of the network device. The **nd_receive** function is part of a CDLI network demuxer. The packet is passed to the **nd_receive** function in mbufs.

The **LANE** device driver passes one packet to the **nd_receive** function at a time.

The device driver sets the **M_BCAST** flag in the **p_mbuf->m_flags** field when a packet is received that has an all-stations broadcast destination address. This address value is defined as FF.FF.FF.FF.FF.FF (hex) for both Token Ring and Ethernet and is defined as C0.00.FF.FF.FF.FF (hex) for Token Ring.

The device driver sets the **M_MCAST** flag in the **p_mbuf->m_flags** field when a packet is received that has a nonindividual address that is different than an all-stations broadcast address.

Any packets received from the network are discarded if they do not fit the currently emulated **LAN** protocol and frame format are discarded.

## Asynchronous Status

When a status event occurs on the device, the **LANE** device driver builds the appropriate status block and calls the **nd_status** function that is specified in the **ndd_t** structure of the network device. The **nd_status** function is part of a CDLI network demuxer.

The following status blocks are defined for the **LANE** device driver:

### *Hard Failure*

When an error occurs within the internal operation of the **ATM LANE** device driver, it is considered unrecoverable. If the device was operational at the time of the error, the **NDD_LIMBO** and **NDD_RUNNING** flags are disabled, and the **NDD_DEAD** flag is set in the **ndd_flags** field, and a hard failure status block is generated.

| Item | Description |
|---|---|
| **code** | Set to NDD_HARD_FAIL |
| **option[0]** | Set to NDD_UCODE_FAIL |

### *Enter Network Recovery Mode*

When the device driver detects an error that requires initiating recovery logic to make the device temporarily unavailable, the following status block is returned by the device driver:

| Item | Description |
|---|---|
| **code** | Set to NDD_LIMBO_ENTER |
| **option[0]** | Set to NDD_UCODE_FAIL |

> **Note:** While the device driver is in this recovery logic, the network connections might not be fully functional. The device driver notifies users when the device is fully functional by way of an NDD_LIMBO_EXIT asynchronous status block.

When a general error occurs during operation of the device, this status block is generated.

### *Exit Network Recovery Mode*

When the device driver has successfully completed recovery logic from the error that made the device temporarily unavailable, the following status block is returned by the device driver. This status block means the device is now fully functional.

| Item | Description |
|---|---|
| **code** | Set to NDD_LIMBO_EXIT |
| **option[0]** | The **option** field is not used. |

## Device Control Operations

The **atmle_ctl** function is used to provide device control functions.

The **atmle_ctl** function provides the following controls:

### *ATMLE_MIB_GET*

This control requests the **LANE** device driver's current ATM LAN Emulation MIB statistics.

The user should pass in the address of an **atmle_mibs_t** structure as defined in **usr/include/sys/atmle_mibs.h**. The driver returns EINVAL if the buffer area is smaller than the required structure.

The **ndd_flags** field can be checked to determine the current state of the LANE device.

### *ATMLE_MIB_QUERY*

This control requests the **LANE** device driver's ATM LAN Emulation MIB support structure.

The user should pass in the address of an **atmle_mibs_t** structure as defined in **usr/include/sys/atmle_mibs.h**. The driver returns EINVAL if the buffer area is smaller than the required structure.

The device driver does *not* support any variables for read_write or write only. If the syntax of a member of the structure is some integer type, the level of support flag is stored in the whole field, regardless of the size of the field. For those fields defined as character arrays, the value is returned only in the first byte in the field.

### *NDD_CLEAR_STATS*

This control requests all the statistics counters kept by the **LANE** device driver to be zeroed.

### *NDD_DISABLE_ADDRESS*

This command disables the receipt of packets destined for a multicast/group address; and for Token Ring, it disables the receipt of packets destined for a functional address.

For Token Ring, the functional address indicator (bit 0, the most significant bit of byte 2) indicates whether the address is a functional address (the bit is a 0) or a group address (the bit is a 1).

In all cases, the **length** field value is required to be 6. Any other value causes the **LANE** device driver to return EINVAL.

*Functional Address*

The reference counts are decremented for those bits in the functional address that are enabled (set to 1). If the reference count for a bit goes to zero, the bit is disabled in the functional address mask for this LE Client.

If no functional addresses are active after receipt of this command, the **TOK_RECEIVE_FUNC** flag in the **ndd_flags** field is reset. If no functional or multicast/group addresses are active after receipt of this command, the **NDD_ALTADDRS** flag in the **ndd_flags** field is reset.

*Multicast/Group Address*
If a multicast/group address that is currently enabled is specified, receipt of packets destined for that group address is disabled. If an address is specified that is not currently enabled, EINVAL is returned.

If no functional or multicast/group addresses are active after receipt of this command, the **NDD_ALTADDRS** flag in the **ndd_flags** field is reset. Additionally for Token Ring, if no multicast/group address is active after receipt of this command, the **TOK_RECEIVE_GROUP** flag in the **ndd_flags** field is reset.

### *NDD_DISABLE_MULTICAST*
The **NDD_DISABLE_MULTICAST** command disables the receipt of *all* packets with unregistered multicast addresses, and only receives those packets whose multicast addresses were registered using the **NDD_ENABLE_ADDRESS** command.

The *arg* and *length* parameters are not used. The **NDD_MULTICAST** flag in the **ndd_flags** field is reset only after the reference count for multicast addresses has reached zero.

### *NDD_ENABLE_ADDRESS*
The **NDD_ENABLE_ADDRESS** command enables the receipt of packets destined for a multicast/group address; and additionally for Token Ring, it enables the receipt of packets destined for a functional address.

For Ethernet, the address is entered in canonical format, which is left-to-right byte order with the I/G (Individual/Group) indicator as the least significant bit of the first byte. For Token Ring, the address format is entered in noncanonical format, which is left-to-right bit and byte order and has a functional address indicator. The functional address indicator (the most significant bit of byte 2) indicates whether the address is a functional address (the bit value is 0) or a group address (the bit value is 1).

In all cases, the **length** field value is required to be 6. Any other length value causes the **LANE** device driver to return EINVAL.

*Functional Address*
The Token-Ring network architecture provides bit-specific functional addresses for widely used functions, such as Ring Parameter Server or Configuration Report Server. Ring stations use functional address masks to identify these functions.

The specified address is OR'ED with the currently specified functional addresses, and the resultant address is set as the functional address for the device. Functional addresses are encoded in a bit-significant format, thereby allowing multiple individual groups to be designated by a single address.

For example, if function G is assigned a functional address of C0.00.00.08.00.00 (hex), and function M is assigned a functional address of C0.00.00.00.00.40 (hex), then ring station Y, whose node contains function G and M, would have a mask of C0.00.00.08.00.40 (hex). Ring station Y would receive packets addressed to either function G or M or to an address like C0.00.00.08.00.48 (hex) because that address contains bits specified in the mask.

**Note:** The **LANE** device driver forces the first 2 bytes of the functional address to be C0.00 (hex). In addition, bits 6 and 7 of byte 5 of the functional address are forced to 0.

The **NDD_ALTADDRS** and **TOK_RECEIVE_FUNC** flags in the **ndd_flags** field are set.

Because functional addresses are encoded in a bit-significant format, reference counts are kept on each of the 31 least significant bits of the address. Reference counts are not kept on the 17 most significant bits (the C0.00 (hex) of the functional address and the functional address indicator bit).

*Multicast/Group Address*
A multicast/group address table is used by the **LANE** device driver to store address filters for incoming multicast/group packets. If the **LANE** device driver is unable to allocate kernel memory when attempting to add a multicast/group address to the table, the address is not added and ENOMEM is returned.

If the **LANE** device driver is successful in adding a multicast/group address, the **NDD_ALTADDRS** flag in the **ndd_flags** field is set. Additionally for Token Ring, the **TOK_RECEIVE_GROUP** flag is set, and the first 2 bytes of the group address are forced to be C0.00 (hex).

### *NDD_ENABLE_MULTICAST*
The **NDD_ENABLE_MULTICAST** command enables the receipt of packets with any multicast (or group) address.

The *arg* and *length* parameters are not used. The **NDD_MULTICAST** flag in the **ndd_flags** field is set.

### *NDD_DEBUG_TRACE*
This control requests a LANE or MPOA driver to toggle the current state of its **debug_trace** configuration flag.

This control is available to the operator through the LANE Ethernet **entstat -t** or LANE Token Ring **tokstat -t** commands, or through the MPOA **mpcstat -t** command. The current state of the **debug_trace** configuration flag is displayed in the output of each command as follows:

- For the **entstat** and **tokstat** commands, NDD_DEBUG_TRACE is enabled only if you see `Driver Flags: Debug`.
- For the **mpcstat** command, you see `Debug Trace: Enabled`.

### *NDD_GET_ALL_STATS*
This control requests all current LANE statistics, based on both the generic LAN statistics and the **ATM LANE** protocol in progress.

For Ethernet, pass in the address of an **ent_ndd_stats_t** structure as defined in the file /**usr/include/sys/cdli_entuser.h**.

For Token Ring, pass in the address of a **tok_ndd_stats_t** structure as defined in the file /**usr/include/sys/cdli_tokuser.h**.

The driver returns EINVAL if the buffer area is smaller than the required structure.

The **ndd_flags** field can be checked to determine the current state of the LANE device.

### *NDD_GET_STATS*
This control requests the current generic LAN statistics based on the **LAN** protocol being emulated.

For Ethernet, pass in the address of an **ent_ndd_stats_t** structure as defined in the file /**usr/include/sys/cdli_entuser.h**.

For Token Ring, pass in the address of a **tok_ndd_stats_t** structure as defined in file /**usr/include/sys/cdli_tokuser.h**.

The **ndd_flags** field can be checked to determine the current state of the LANE device.

### *NDD_MIB_ADDR*
This control requests the current receive addresses that are enabled on the **LANE** device driver.

The following address types are returned, up to the amount of memory specified to accept the address list:

- Local LAN MAC Address
- Broadcast Address FF.FF.FF.FF.FF.FF (hex)
- Broadcast Address C0.00.FF.FF.FF.FF (hex)
- (returned for Token Ring only)
- Functional Address Mask
- (returned for Token Ring only, and only if at least one functional address has been enabled)
- Multicast/Group Address 1 through n
- (returned only if at least one multicast/group address has been enabled)

Each address is 6-bytes in length.

### *NDD_MIB_GET*

This control requests the current MIB statistics based on whether the LAN being emulated is Ethernet or Token Ring.

If Ethernet, pass in the address of an **ethernet_all_mib_t** structure as defined in the file **/usr/include/sys/ethernet_mibs.h**.

If Token Ring, pass in the address of a **token_ring_all_mib_t** structure as defined in the file **/usr/include/sys/tokenring_mibs.h**.

The driver returns EINVAL if the buffer area is smaller than the required structure.

The **ndd_flags** field can be checked to determine the current state of the LANE device.

### *NDD_MIB_QUERY*

This control requests **LANE** device driver's MIB support structure based on whether the LAN being emulated is Ethernet or Token Ring.

If Ethernet, pass in the address of an **ethernet_all_mib_t** structure as defined in the file **/usr/include/sys/ethernet_mibs.h**.

If Token Ring, pass in the address of a **token_ring_all_mib_t** structure as defined in the file **/usr/include/sys/tokenring_mibs.h**.

The driver returns EINVAL if the buffer area is smaller than the required structure.

The device driver does *not* support any variables for read_write or write only. If the syntax of a member of the structure is some integer type, the level of support flag is stored in the whole field, regardless of the size of the field. For those fields which are defined as character arrays, the value is returned only in the first byte in the field.

## Tracing and Error Logging in the ATM LANE Device Driver

The **LANE** device driver has two trace points:

- 3A1 - Normal Code Paths
- 3A2 - Error Conditions

Tracing can be enabled through SMIT or with the **trace** command.

```
trace -a -j 3a1,3a2
```

Tracing can be disabled through SMIT or with the **trcstop** command. After trace is stopped, the results can be formatted into readable text with the **trcrpt** command.

```
trcrpt > /tmp/trc.out
```

### *LANE error log templates:*

The LANE error logs contain the following error identifiers and descriptions.

| Item | Description |
|---|---|
| **ERRID_ATMLE_MEM_ERR** | An error occurred while attempting to allocate memory or pin the code. This error log entry accompanies return code ENOMEM on an open or control operation. |

| Item | Description |
| --- | --- |
| **ERRID_ATMLE_LOST_SW** | The **LANE** device driver lost contact with the ATM switch. The device driver enters Network Recovery Mode in an attempt to recover from the error and is temporarily unavailable during the recovery procedure. This generally occurs when the cable is unplugged from the switch or ATM adapter. |
| **ERRID_ATMLE_REGAIN_SW** | Contact with the ATM switch has been re-established (for example, the cable has been plugged back in). |
| **ERRID_ATMLE_NET_FAIL** | The device driver has gone into Network Recovery Mode in an attempt to recover from a network error and is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists. |
| **ERRID_ATMLE_RCVRY_CMPLETE** | The network error that caused the **LANE** device driver to go into error recovery mode has been corrected. |

## Adding an ATM MPOA Client

A Multi-Protocol Over ATM (MPOA) Client (MPC) can be added to the system to allow ATM LANE packets that would normally be routed through various LANE IP Subnets or Logical IP Subnets (LISs) within an ATM network, to be sent and received over shortcut paths that do not contain routers.

MPOA can provide significant savings on end-to-end throughput performance for large data transfers, and can free up resources in routers that might otherwise be used up handling packets that could have bypassed routers altogether.

Only one MPOA Client is established per node. This MPC can support multiple ATM ports, containing LE Clients/Servers and MPOA Servers. The key requirement being, that for this MPC to create shortcut paths, each remote target node must also support MPOA Client, and must be directly accessible using the matrix of switches representing the ATM network.

A user with root authority can add this MPOA Client using the **smit mpoa_panel** fast path, or click **Devices —> Communication —> ATM Adapter —> Services —> Multi-Protocol Over ATM (MPOA)**.

No configuration entries are required for the MPOA Client. Ease-of-use default values are provided for each of the attributes derived from ATM Forum recommendations.

Configuration help text is also available within MPOA Client SMIT to aid in making any modifications to attribute default values.

## Configuration Parameters for ATM MPOA Client

The **ATM LANE** device driver supports the following configuration parameters for the MPOA Client:

| Item | Description |
| --- | --- |
| ***auto_cfg*** | Auto Configuration with LEC/LECS. Specifies whether the MPOA Client is to be automatically configured using LANE Configuration Server (LECS). Select **Yes** if a primary LE Client is used to obtain the MPOA configuration attributes, which overrides any manual or default values. The default value is No (manual configuration). The attribute values are: **Yes** - auto configuration **No** - manual configuration |

| Item | Description |
|---|---|
| ***debug_trace*** | Specifies whether this MPOA Client should keep a real time debug log within the kernel and allow full system trace capability. Select **Yes** to enable full tracing capabilities for this MPOA Client. Select **No** for optimal performance when minimal tracing is desired. The default is **Yes** (full tracing capability). |
| ***fragment*** | Enables MPOA fragmentation and specifies whether fragmentation should be performed on packets that exceed the maximum transmission unit (MTU) returned in the MPOA Resolution Reply. Select **Yes** to have outgoing packets fragmented as needed. Select **No** to avoid having outgoing packets fragmented. Selecting No causes outgoing packets to be sent down the LANE path when fragmentation must be performed. Incoming packets are always fragmented as needed even if No has been selected. The default value is **Yes**. |
| ***hold_down_time*** | Failed resolution request retry Hold Down Time (in seconds). Specifies the length of time to wait before reinitiating a failed address resolution attempt. This value is normally set to a value greater than *retry_time_max*. This attribute correlates to ATM Forum MPC Configuration parameter *MPC-p6*. The default value is 160 seconds. |
| ***init_retry_time*** | Initial Request Retry Time (in seconds). Specifies the length of time to wait before sending the first retry of a request that does not receive a response. This attribute correlates to ATM Forum MPC Configuration parameter *MPC-p4*. The default value is 5 seconds. |
| ***retry_time_max*** | Maximum Request Retry Time (in seconds). Specifies the maximum length of time to wait when retrying requests that have not received a response. Each retry duration after the initial retry are doubled (2x) until the retry duration reaches this Maximum Request Retry Time. All subsequent retries wait this maximum value. This attribute correlates to ATM Forum MPC Configuration parameter *MPC-p5*. The default value is 40 seconds. |
| ***sc_setup_count*** | Shortcut Setup Frame Count. This attribute is used in conjunction with *sc_setup_time* to determine when to establish a shortcut path. After the MPC has forwarded at least *sc_setup_count* packets to the same target within a period of *sc_setup_time*, the MPC attempts to create a shortcut VCC. This attribute correlates to ATM Forum MPC Configuration parameter *MPC-p1*. The default value is 10 packets. |
| ***sc_setup_time*** | Shortcut Setup Frame Time (in seconds). This attribute is used in conjunction with *sc_setup_count* above to determine when to establish a shortcut path. After the MPC has forwarded at least *sc_setup_count* packets to the same target within a period of *sc_setup_time*, the MPC attempts to create a shortcut VCC. This attribute correlates to ATM Forum MPC Configuration parameter *MPC-p2*. The default value is 1 second. |
| ***vcc_inact_time*** | VCC Inactivity Timeout value (in minutes). Specifies the maximum length of time to keep a shortcut VCC enabled when there is no send or receive activity on that VCC. The default value is 20 minutes. |

## Tracing and Error Logging in the ATM MPOA Client

The ATM MPOA Client has two trace points:

- 3A3 - Normal Code Paths
- 3A4 - Error Conditions

Tracing can be enabled through SMIT or with the **trace** command.

```
trace -a -j 3a3,3a4
```

Tracing can be disabled through SMIT or with the **trcstop** command. After trace is stopped, the results can be formatted into readable text with the **trcrpt** command.

```
trcrpt > /tmp/trc.out
```

### *MPOA Client error log templates*

Each of the MPOA Client error log templates are prefixed with **ERRID_MPOA**.

An example of an MPOA error entry is as follows:

**ERRID_MPOA_MEM_ERR**
An error occurred while attempting to allocate kernel memory.

## Getting Client Status

Three commands are available to obtain status information related to ATM **LANE** clients.

- The **entstat** command and **tokstat** command are used to obtain general ethernet or tokenring device status.
- The **lecstat** command is used to obtain more specific information about a **LANE** client.
- The **mpcstat** command is used to obtain MPOA client status information.

**Related information**

entstat Command
lecstat Command
mpcstat Command
tokstat Command

# PCI Token-Ring Device Drivers

The Token-Ring device drivers that are listed in this section are dynamically loadable. The device driver is automatically loaded into the system at device configuration time as part of the configuration process.

- PCI Token-Ring High PerformanceDevice Driver (14101800)
- PCI Token-Ring Device Driver (14103e00)

The interface to the device is through the kernel services known as *Network Services*. Interfacing to the device driver is achieved by calling the device driver's entry points to perform the following actions:

- Opening the device
- Closing the device
- Transmitting data
- Issuing device control commands

The PCI Token-Ring High Performance Device Driver (14101800) interfaces with the PCI Token-Ring High-Performance Network Adapter (14101800). The adapter is IEEE 802.5 compatible and supports both 4 and 16 Mbps networks. The adapter supports only an RJ-45 connection.

The PCI Token-Ring Device Driver (14103e00) interfaces with the PCI Token-Ring Network Adapter (14103e00). The adapter is IEEE 802.5 compatible and supports both 4 and 16 Mbps networks. The adapter supports both an RJ-45 and a 9 Pin connection.

## Configuration Parameters

The configuration parameter that is explained in this section is supported by all PCI token-ring device drivers.

**Ring Speed**
The device driver supports a user-configurable parameter that indicates if the token-ring is to run at 4 or 16 Mbps.

The device driver supports a user-configurable parameter that selects the ring speed of the adapter. There are three options for the ring speed: 4, 16, or autosense.

1. If 4 is selected, the device driver opens the adapter with 4 Mbits. It returns an error if the ring speed does not match the network speed.
2. If 16 is selected, the device driver opens the adapter with 16 Mbits. It returns an error if the ring speed does not match the network speed.
3. If autosense is selected, the adapter guarantees a successful open, and the speed used to open is dependent on the following:
   - If the adapter is opened on an existing network the speed is determined by the ring speed of the network.
   - If the device is opened on a new network and the adapter is new, 16 Mbits is used. Or, if the adapter opened successfully, the ring speed is determined by the speed of the last successful open.

**Software Transmit Queue**
The device driver supports a user-configurable transmit queue that can be set to store between 32 and 2048 transmit request pointers. Each transmit request pointer corresponds to a transmit request that might be for several buffers of data.

**Receive Queue**
The device driver supports a user-configurable receive queue that can be set to store between 32 and 160 receive buffers. These buffers are **mbuf** clusters into which the device writes the received data.

**Full Duplex**
Indicates whether the adapter is operating in full-duplex or half-duplex mode. If this field is set to `yes`, the device driver programs the adapter to be in full-duplex mode. The default value is half-duplex.

**Attention MAC Frames**
The device driver supports a user-configurable parameter that indicates if attention MAC frames should be received.

**Beacon MAC Frames**
The device driver supports a user-configurable parameter that indicates if beacon MAC frames should be received.

**Network Address**
The driver supports the use of the device's hardware address as the network address or an alternate network address configured through software. When an alternate address is used, any valid individual address can be used. The most significant bit of the address must be set to zero.

In addition, the following configuration parameters are supported by the PCI Token-Ring High Performance Device Driver (14101800):

**Priority Data Transmission**
The device driver supports a user option to request priority transmission of the data packets.

**Software Priority Transmit Queue**
The device driver supports a user-configurable priority transmit queue that can be set to store between 32 and 160 transmit request pointers. Each transmit request pointer corresponds to a transmit request that might be for several buffers of data.

## Device Driver Configuration and Unconfiguration

The configuration entry points of the device drivers conform to the guidelines for kernel object file entry points.

These configuration entry points are as follows:

- **tok_config** for the PCI Token-Ring High Performance Device Driver (14101800).
- **cs_config** for the PCI Token-Ring Device Driver (14103e00).

## Device Driver Open

The Token-Ring device driver performs a synchronous open. The device is initialized at this time. When the resources are successfully allocated, the device starts the process of attaching the device to the network.

If the connection is successful, the **NDD_RUNNING** flag is set in the `ndd_flags` field, and an NDD_CONNECTED status block is sent.

If the device connection fails, the **NDD_LIMBO** flag is set in the `ndd_flags` field, and an NDD_LIMBO_ENTRY status block is sent.

If the device is eventually connected, the **NDD_LIMBO** flag is turned off, and the **NDD_RUNNING** flag is set in the `ndd_flags` field. Both NDD_CONNECTED and NDD_LIMBO_EXIT status blocks are set.

The entry points are as follows:

- **tok_open** for the PCI Token-Ring High Performance Device Driver (14101800).
- **cs_open** for the PCI Token-Ring Device Driver (14103e00).

## Device Driver Close

This function resets the device to a known state and frees system resources associated with the device.

The device is not detached from the network until the device's transmit queue is allowed to drain.

The close entry points are as follows:

- **tok_close** for the PCI Token-Ring High Performance Device Driver (14101800).
- **cs_close** for the PCI Token-Ring Device Driver (14103e00).

## Data Transmission

The device drivers do not support **mbuf** structures from user memory that have the **M_EXT** flag set.

If the destination address in the packet is a broadcast address, the **M_BCAST** flag in the `p_mbuf->m_flags` field must be set prior to entering this routine. A broadcast address is defined as 0xFFFF FFFF FFFF or 0xC000 FFFF FFFF. If the destination address in the packet is a multicast address, the **M_MCAST** flag in the `p_mbuf->m_flags` field must be set prior to entering this routine. A multicast address is defined as a non-individual address other than a broadcast address. The device driver keeps statistics based on the **M_BCAST** and **M_MCAST** flags.

If a packet is transmitted with a destination address that matches the adapter's address, the packet is received. This is true for the adapter's physical address, broadcast addresses (0xC000 FFFF FFFF or 0xFFFF FFFF FFFF), enabled functional addresses, or an enabled group address.

The output entry points are as follows:

- **tok_output** for the PCI Token-Ring High Performance Device Driver (14101800).
- **cs_close** for the PCI Token-Ring Device Driver (14103e00).

## Data Reception

When the Token-Ring device driver receives a valid packet from the network device, the Token-Ring device driver calls the **nd_receive()** function specified in the **ndd_t** structure of the network device. The **nd_receive()** function is part of a CDLI network demuxer.

The packet is passed to the **nd_receive()** function in the **mbuf** structures.

The Token-Ring device driver passes only one packet to the **nd_receive()** function at a time.

The device driver sets the **M_BCAST** flag in the `p_mbuf->m_flags` field when a packet that has an all-stations broadcast address is received. This address is defined as 0xFFFF FFFF FFFF or 0xC000 FFFF FFFF.

The device driver sets the **M_MCAST** flag in the `p_mbuf->m_flags` field when a packet is received that has a non-individual address that is different from the all-stations broadcast address.

The adapter does not pass invalid packets to the device driver.

## Asynchronous Status

When a status event occurs on the device, the Token-Ring device driver builds the appropriate status block and calls the **nd_status()** function specified in the **ndd_t** structure of the network device. The **nd_status()** function is part of a CDLI network demuxer.

The following status blocks are defined for the Token-Ring device driver:

### *Hard Failure*

When a hard failure occurs on the Token-Ring device, the following status blocks are returned by the Token-Ring device driver.

One of these status blocks indicates that a fatal error has occurred.

**NDD_HARD_FAIL**
When a transmit error occurs, it tries to recover. If the error is unrecoverable, this status block is generated.

**code**
Set to NDD_HARD_FAIL.

**option[0]**
Set to NDD_HARD_FAIL.

**option[ ]**
The remainder of the status block can be used to return additional status information.

### *Enter Network Recovery Mode*

When the device driver detects an error that requires initiating recovery logic to make the device temporarily unavailable, the following status block is returned by the device driver.

**Note:** While the device driver is in this recovery logic, the device might not be fully functional. The device driver notifies users when the device is fully functional by way of an NDD_LIMBO_EXIT asynchronous status block:

| Item | Description |
|---|---|
| **code** | Set to NDD_LIMBO_ENTER. |

| Item | Description |
|---|---|
| **option[0]** | Set to one of the following:<br>• NDD_CMD_FAIL<br>• NDD_ADAP_CHECK<br>• NDD_TX_ERR<br>• NDD_TX_TIMEOUT<br>• NDD_AUTO_RMV<br>• TOK_ADAP_OPEN<br>• TOK_ADAP_INIT<br>• TOK_DMA_FAIL<br>• TOK_RING_SPEED<br>• TOK_RMV_ADAP<br>• TOK_WIRE_FAULT |
| **option[ ]** | The remainder of the status block can be used to return additional status information by the device driver. |

### *Exit Network Recovery Mode*

When the device driver has successfully completed recovery logic from the error that made the device temporarily unavailable, the following status block is returned by the device driver:

| Item | Description |
|---|---|
| **code** | Set to NDD_LIMBO_EXIT. |
| **option[ ]** | The option fields are not used. |

The device is now fully functional.

## Device Control Operations

The **ndd_ctl** entry point is used to provide device control functions.

**NDD_GET_STATS**
The user should pass in the **tok_ndd_stats_t** structure as defined in the **sys/cdli_tokuser.h** file. The driver fails a call with a buffer smaller than the structure.

The structure must be in kernel heap so that the device driver can copy the statistics into it. Also, it must be pinned.

**NDD_PROMISCUOUS_ON**
Setting promiscuous mode will *not* cause non-LLC frames to be received by the driver unless the user also enables those filters (Attention MAC frames, Beacon MAC frames).

The driver maintains a counter of requests.

**NDD_PROMISCUOUS_OFF**
This command releases a request from a user to **PROMISCUOUS_ON**; it will not exit the mode on the adapter if more requests are outstanding.

**NDD_MIB_QUERY**
The **arg** parameter specifies the address of the **token_ring_all_mib_t** structure. This structure is defined in the **/usr/include/sys/tokenring_mibs.h** file.

The device driver does *not* support any variables for read_write or write only. If the syntax of a member of the structure is an integer type, the level of support flag is stored in the whole field, regardless of the size of the field. For those fields that are defined as character arrays, the value is returned only in the first byte in the field.

**NDD_MIB_GET**
The **arg** parameter specifies the address of the **token_ring_all_mib_t** structure. This structure is defined in the **/usr/include/sys/tokenring_mibs.h** file.

**NDD_ENABLE_ADDRESS**
This command enables the receipt of packets with a functional or a group address. The functional address indicator (bit 0 "the MSB" of byte 2) indicates whether the address is a functional address (bit 0) or a group address (bit 1). The `length` field is not used because the address must be 6 bytes in length.

**functional address**
The specified address is ORed with the currently specified functional addresses, and the resultant address is set as the functional address for the device. Functional addresses are encoded in a bit-significant format, thereby allowing multiple individual groups to be designated by a single address.

The Token-Ring network architecture provides bit-specific functional addresses for widely used functions, such as configuration report server. Ring stations use functional address "masks" to identify these functions. For example, if function G is assigned a functional address of 0xC000 0008 0000, and function M is assigned a function address of 0xC000 0000 0040, then ring station Y, whose node contains function G and M, would have a mask of 0xC000 0008 0040. Ring station Y would receive packets addressed to either function G or M or to an address, such as 0xC000 0008 0048, because that address contains bits specified in the "mask."

The **NDD_ALTADDRS** and **TOK_RECEIVE_FUNC** flags in the `ndd_flags` field are set.

Because functional addresses are encoded in a bit-significant format, reference counts are kept on each of the 31 least significant bits of the address.

**group address**
The device supports 256 general group addresses. The promiscuous mode is turned on when the group addresses to be set is more than 256. The device driver maintains a reference count on this operation.

The device supports 256 general group addresses. The promiscuous mode is turned on when the group address needed to be set are more than 256. The device driver will maintain a reference count on this operation.

The **NDD_ALTADDRS** and **TOK_RECEIVE_GROUP** flags in the `ndd_flags` field are set.

**NDD_DISABLE_ADDRESS**
This command disables the receipt of packets with a functional or a group address. The functional address indicator (bit 0 "the MSB" of byte 2) indicates whether the address is a functional address (bit 0) or a group address (bit 1). The `length` field is not used because the address must be 6 bytes in length.

**functional address**
The reference counts are decremented for those bits in the functional address that are 1 (on). If the reference count for a bit goes to 0, the bit is "turned off" in the functional address for the device.

If no functional addresses are active after receipt of this command, the **TOK_RECEIVE_FUNC** flag in the `ndd_flags` field is reset. If no functional or group addresses are active after receipt of this command, the **NDD_ALTADDRS** flag in the `ndd_flags` field is reset.

**group address**
If group address enable is less than 256, the driver sends a command to the device to disable the receipt of the packets with the specified group address. Otherwise, the group address is deleted from the group address table.

If there are less than 256 group addresses enabled after the receipt of this command, the promiscuous mode is turned off.

If no group address is active after receipt of this command, the **TOK_RECEIVE_GROUP** flag in the `ndd_flags` field is reset. If no functional or group addresses are active after receipt of this command, the **NDD_ALTADDRS** flag in the `ndd_flags` field is reset.

**NDD_PRIORITY_ADDRESS**
The driver returns the address of the device driver's priority transmit routine.

**NDD_MIB_ADDR**
The driver returns at least three addresses that are device physical addresses (or alternate addresses specified by the user), two broadcast addresses (0xFFFFFFFFFFFF and 0xC000 FFFF FFFF), and any additional addresses specified by the user, such as functional addresses and group addresses.

**NDD_CLEAR_STATS**
The counters kept by the device are zeroed.

**NDD_GET_ALL_STATS**
Used to gather all statistics for the specified device. The **arg** parameter specifies the address of the statistics structure for this particular device type. The folowing structures are available:

- The **sky_all_stats_t** structure is available for the PCI Token-Ring High Performance Device Driver (14101800), and is defined in the device-specific **/usr/include/sys/cdli_tokuser.h** include file.
- The **cs_all_stats_t** structure is available for the PCI Token-Ring Device Driver (14103e00), and is defined in the device-specific **/usr/include/sys/cdli_tokuser.cstok.h** include file.

The statistics that are returned contain information obtained from the device. If the device is inoperable, the statistics returned are not the current device statistics. The copy of the `ndd_flags` field can be checked to determine the state of the device.

## Reliability, Availability, and Serviceability (RAS)

The following sections explain the trace points and error logging for PCI Token-Ring device drivers.

### *Trace*

For LAN device drivers, trace points enable error monitoring as well as tracking packets as they move through the driver. The drivers issue trace points for some or all of the following conditions:

- Beginning and ending of main functions in the main path
- Error conditions
- Beginning and ending of each function that is tracking buffers outside of the main path
- Debugging purposes (These trace points are only enabled when the driver is compiled with the **-DDEBUG** option turned, therefore, the driver can contain as many of these trace points as needed.)

Following is a list of trace hooks and location of definition files for the existing ethernet device drivers.

*The PCI Token-Ring High Performance Device Driver (14101800)*
The definition file and trace hook IDs for PCI Token-Ring High Performance device driver (14101800) are as follows:

**Definition File: /sys/cdli_tokuser.h**

**Trace Hook IDs**

- Transmit 2A7
- Receive 2A8
- Error 2A9
- Other 2AA

*The PCI Token-Ring (14103e00) Device Driver*
Following is a list of trace hooks and location of definition files for the existing ethernet device drivers.

**Definition File: /sys/cdli_tokuser.cstok.h**

**Trace Hook IDs**

- Transmit 2DA
- Receive 2DB
- General 2DC

## *Error Logging*

Following are the error IDs for the PCI Token-Ring High Performance Device Driver and the PCI Token-Ring Device Driver.

*PCI Token-Ring High Performance Device Driver (14101800)*

The error IDs for the PCI Token-Ring High Performance Device Driver (14101800) are as follows:

**ERRID_STOK_ADAP_CHECK**
The microcode on the device performs a series of diagnostic checks when the device is idle. These checks can find errors, and they are reported as adapter checks. If the device is connected to the network when this error occurs, the device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_STOK_ADAP_OPEN**
Enables the device driver to open the device. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_STOK_AUTO_RMV**
An internal hardware error following the beacon automatic removal process was detected. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_STOK_RING_SPEED**
The ring speed (or ring data rate) is probably wrong. Contact the network administrator to determine the speed of the ring. The device driver only retries twice at 2-minute intervals after this error log entry is generated.

**ERRID_STOK_DMAFAIL**
The device detected a DMA error in a TX or RX operation. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.

**ERRID_STOK_BUS_ERR**
The device detected a Micro Channel bus error. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**Note:** Micro Channel is only supported on AIX® 5.1 and earlier.

**ERRID_STOK_DUP_ADDR**
The device detected that another station on the ring has a device address that is the same as the device address being tested. Contact the network administrator to determine why.

**ERRID_STOK_MEM_ERR**
An error occurred while allocating memory or timer control block structures.

**ERRID_STOK_RCVRY_EXIT**
The error that caused the device driver to go into error recovery mode was corrected.

**ERRID_STOK_RMV_ADAP**
The device received a remove ring station MAC frame indicating that a network management function directed this device to get off the ring. Contact the network administrator to determine why.

**ERRID_STOK_WIRE_FAULT**
There is a loose (or bad) cable between the device and the MAU. There is a chance that it might be a bad device. The device driver goes into Network Recover Mode in an attempt to recover from the error.

The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_STOK_TX_TIMEOUT**
The transmit watchdog timer expired before transmitting a frame. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_STOK_CTL_ERR**
The ioctl watchdog timer expired before the device driver received a response from the device. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

*PCI Token-Ring Device Driver (14103e00)*
The error IDs for the PCI Token-Ring Device Driver (14103e00) are as follows:

**ERRID_CSTOK_ADAP_CHECK**
The microcode on the device performs a series of diagnostic checks when the device is idle on initialization. These checks find errors and they are reported as adapter checks. If the device was connected to the network when this error occurred, the device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. After this error log entry has been generated, the device driver will retry 3 times with no delay between retries. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_ADAP_OPEN**
The device driver was unable to open the device. The device driver will go into Network Recovery Mode in an attempt to recover from this error. The device is temporarily unavailable during the recovery procedure. The device driver will retry indefinitely with a 30 second delay between retries to recover. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_AUTO_RMV**
An internal hardware error following the beacon automatic removal process has been detected. The device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_RING_SPEED**
The ring speed or ring data rate is probably wrong. Contact the network administrator to determine the speed of the ring. The device driver will only retry twice at 2 minute intervals after this error log entry has been generated.

**ERRID_CSTOK_DMAFAIL**
The device detected a DMA error in a TX or RX operation. The device driver will go into Network Recovery Mode in an attempt to recover from this error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_BUS_ERR**
The device detected a PCI bus error. The device driver will go into Network Recovery Mode in an attempt to recover from this error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_DUP_ADDR**
The device has detected that another station on the ring has a device address which is the same as the device address being tested. Contact network administrator to determine why.

**ERRID_CSTOK_MEM_ERR**
An error occurred while allocating memory or timer control block structures. This usually implies the sytem has run out of available memory. User intervention is required.

**ERRID_CSTOK_RCVRY_ENTER**
An error has occurred which caused the device driver to go into network recovery.

**ERRID_CSTOK_RCVRY_EXIT**
The error which caused the device driver to go into Network Recovery Mode has been corrected.

**ERRID_CSTOK_RMV_ADAP**
The device has received a remove ring station MAC frame indicating that a network management function has directed this device to get off the ring. The device driver will only twice with 6 minute delay between retries after this error log entry has been generated. Contact network administrator to determine why.

**ERRID_CSTOK_WIRE_FAULT**
There is probably a loose ( or bad ) cable between the device and the MAU. There is some chance that it might be a bad device. The device driver will go into Network Recovery Mode in an attempt to recover from this error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_RX_ERR**
The device has detected a receive error. The device driver will go into Network Recovery Mode in an attempt to recover from this error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_TX_ERR**
The device has detected a transmit error. The device driver will go into Network Recovery Mode in an attempt to recover from this error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_TX_TMOUT**
The transmit watchdog timer has expired before the transmit of a frame has completed. The device driver will go into Network Recovery Mode in an attempt to recover from this error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_CMD_TMOUT**
The ioctl watchdog timer has expired before the device driver received a response from the device. The device driver will go into Network Recovery Mode in an attempt to recover from this error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

**ERRID_CSTOK_PIO_ERR**
The driver has encountered a PIO operation error. The device driver will attempt to retry the operation 3 times before it will fail the command and return in the DEAD state to the user. User intervention is required.

**ERRID_CSTOK_PERM_HW**
The microcode on the device performs a series of diagnostic checks on initialization. These checks can find errors and they are reported as adapter checks. If the error occurs 4 times during adapter initialization this error log will be generated and the device considered inoperable. User intervention is required.

**ERRID_CSTOK_ASB_ERR**
The adapter has indicated that the processing of a TokenRing mac command failed.

**ERRID_CSTOK_AUTO_FAIL**
The ring speed of the adapter is set to autosense, and open has failed because this adapter is the only one on the ring. User intervention is required.

**ERRID_CSTOK_EISR**
If the adapter detects a PCI Master or Target Abort, the Error Interrupt Status Register (EISR) will be set.

**ERRID_CSTOK_CMD_ERR**
Adapter failed command due to a transient error and goes into limbo one time, if that fails the adapter goes into the dead state.

**ERRID_CSTOK_EEH_ENTER**
The adapter encountered a Bus I/O Error, and is attempting to recover by using the EEH recovery process.

**ERRID_CSTOK_EEH_EXIT**
The adapter sucessfully recovered from the I/O Error by using the EEH recovery process.

**ERRID_CSTOK_EEH_HW_ERR**
The adapter could not recover from the EEH error. The EEH error was the result of an adapter error, and not a bus error (logged by the kernel).

## Ethernet Device Drivers

The Ethernet device drivers that are listed in this section are dynamically loadable. The device drivers are automatically loaded into the system at device configuration time as part of the configuration process.

- PCI Ethernet Adapter Device Driver (22100020)
- 10/100Mbps Ethernet PCI Adapter Device Driver (23100020)
- 10/100Mbps Ethernet PCI Adapter II Device Driver (1410ff01)
- Gigabit Ethernet-SX PCI Adapter Device Driver (14100401)
- Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802)
- 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902)
- 2-Port Gigabit Ethernet-SX PCI-X Adapter (14108802)
- 2-Port 10/100/1000 Base-TX PCI-X Adapter (14108902)
- 2-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4140909)
- 2-Port Gigabit Ethernet PCI-Express Combo Adapter (e4143a161410a003)
- 2-Port Integrated Gigabit Ethernet PCI-Express Adapter (e4143a161410ed03)
- 2-port 10 Gigabit Ethernet SR PCIe2 Adapter (a21910071410d003)
- 4-Port Gigabit Ethernet PCI-Express Adapter (e414571614102004)
- 4-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4143009)
- 1 Gigabit Ethernet 4-port Mezzanine Adapter (e4145616e4140518) and 1GbE 4-port Mezzanine Adapter (e4145616e4140528)
- 10 Gigabit Ethernet 4-port Mezzanine PCIe Adapter (a2191007df1033e7)
- 10 Gigabit Ethernet-SR PCI-X Adapter (1410ba02)
- 10 Gigabit Ethernet-LR PCI_X Adapter (1410bb02)
- 10 Gigabit Ethernet PCI Express Dual Port Adapter (7710008077108001)
- 10 Gigabit Ethernet-SR PCI Express Dual Port Adapter (771000801410b003)
- 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter (1410eb02)
- 10 Gigabit Ethernet-LR PCI-X 2.0 DDR Adapter (1410ec02)
- Gigabit Ethernet-SX Adapter (e414a816)
- Gigabit Ethernet-SX Adapter (14101403)
- Gigabit Ethernet-SX Adapter (14106703)
- 4-Port 10/100/1000 Base-TX PCI-X Adapter (14101103)
- 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)
- 2-Port Gigabit Ethernet-SX PCI-Express Adapter (14103f03)
- 2-Port 10/100/1000 Base-TX PCI-Express Adapter (14104003)
- Host Ethernet Adapter Device Driver
- Int Multifunction Card Copper SFP+ 10 Gigabit Ethernet (a219100714100a04) and Base-TX 10/100/1000 1 Gigabit Ethernet (a21910071410d203)
- Int Multifunction Card SR Optical 10 Gigabit Ethernet (a219100714100904) and Base-TX 10/100/1000 1 Gigabit Ethernet (a21910071410d203)
- PCIe2 2-port 10 Gigabit Ethernet SFP+Copper Adapter (a21910071410d103)

For each Ethernet device, the interface to the device driver is achieved by calling the entry points for opening, closing, transmitting data, and issuing device control commands.

There are a number of Ethernet device drivers in use. All drivers provide PCI-based connections to an Ethernet network, and support both Standard and IEEE 802.3 Ethernet Protocols.

The PCI Ethernet Adapter Device Driver (22100020) supports the PCI Ethernet BNC/RJ-45 Adapter (feature 2985) and the PCI Ethernet BNC/AUI Adapter (feature 2987) and the integrated Ethernet port on certain systems.

The 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020) supports the 10/100 Mbps Ethernet PCI Adapter (feature 2968) and the Four Port 10/100 Mbps Ethernet PCI Adapter (features 4951 and 4961) and the integrated Ethernet port on certain systems.

The 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) supports the 10/100 Mbps Ethernet PCI Adapter II (feature 4962) and the integrated Ethernet port on certain systems.

The Gigabit Ethernet-SX PCI Adapter Device Driver (14100401) supports the Gigabit Ethernet-SX PCI Adapter (feature 2969) and the 10/100/1000 Base-T Ethernet Adapter (feature 2975).

The Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802) supports the Gigabit Ethernet-SX PCI-X Adapter (feature 5700).

The 10/100/1000 Base-TX Ethernet PCI-X Adapter Device Driver (14106902) supports the 10/100/1000 Base-TX Ethernet PCI-X Adapter (feature 5701).

The 2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802) supports the 2-Port Gigabit Ethernet-SX PCI-X Adapter (feature 5707).

The 2-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14108902) supports the 2-Port 10/100/1000 Base-TX PCI-X Adapter (feature 5706).

The 2-Port Integrated Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a161410ed03) supports the Gigabit Ethernet PCI-Express Adapter (feature 5709).

The 2-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a16e4140909) supports the Gigabit Ethernet PCI-Express Adapter (feature 8243).

The 2-Port Gigabit Ethernet PCI-Express Combo Adapter Device Driver (e4143a161410a003) supports the Gigabit Ethernet PCI-Express Adapter (feature 8725).

The 2-port 10 Gigabit Ethernet SR PCIe2 Adapter Device Driver (a21910071410d003) supports the 10 Gigabit Ethernet PCI-Express Adapter (feature 5287).

The 4-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a16e4143009) supports the Gigabit Ethernet PCI-Express Adapter (feature 8291).

The PCIe2 2-port 10 Gigabit Ethernet SFP+Copper Adapter Device Driver (a21910071410d103) supports the 10 Gigabit Ethernet PCI-Express Adapter (feature 5287).

The 1 Ethernet 4-port Mezzanine Adapter Device Driver (e4145616e4140518) and 1 Ethernet 4-port Mezzanine Adapter Device Driver (e4145616e4140528) supports the Gigabit Ethernet PCI-Express Adapter (feature 1763).

The 10 Gigabit Ethernet Ethernet PCI Express Dual Port Adapter Device Driver (7710008077108001) supports the 10 Gigabit Ethernet PCI-Express Adapter (feature 8275).

The 10 Gigabit Ethernet-SR PCI-X Adapter Device Driver (1410ba02) supports the 10 Gigabit Ethernet-SR PCI-X Adapter (feature 5718).

The 10 Gigabit Ethernet-LR PCI-X Adapter Device Driver (1410bb02) supports the 10 Gigabit Ethernet-LR PCI-X Adapter (feature 5719).

The 10 Gigabit Ethernet-SR PCI Express Dual Port Adapter Device Driver (771000801410b003) supports the 10 Gigabit Ethernet PCI-Express Adapter (feature 5708).

The 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter Device Driver supports the 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter (feature 5721).

The 10 Gigabit Ethernet-LR PCI-X 2.0 DDR Adapter Device Driver supports the 10 Gigabit Ethernet-LR PCI-X 2.0 DDR Adapter (feature 5722).

The 10 Gigabit Ethernet 4-port Mezzanine Adapter Device Driver (a2191007df1033e7) supports the 10 Gigabit Ethernet PCI-Express Adapter (feature 1762).

The Gigabit Ethernet-SX Adapter Device Driver (e414a816) supports the eServer™ BladeCenter JS20 Gigabit Ethernet-SX Adapter.

The Gigabit Ethernet-SX Adapter Device Driver (14101403) supports the eServer™ BladeCenter JS21 Gigabit Ethernet-SX Adapter.

The Gigabit Ethernet-SX Adapter Device Driver (14106703) supports the eServer™ BladeCenter Multiple Switch Interface Module Gigabit Ethernet-SX Adapter.

The 4-Port 10/100/1000 Base-TX Ethernet PCI-X Adapter Device Driver (14101103) supports the 4-Port 10/100/1000 Base-TX PCI-X Adapter (feature 5740).

The 4-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14106803) supports the 4-Port 10/100/1000 Ethernet PCI-E Adapter (feature 5717).

The 2-Port Gigabit Ethernet-SX PCI-Express Adapter Device Driver (14103f03) supports the 2-Port Gigabit Ethernet-SX PCI-Express Adapter (feature 5768).

The 2-Port 10/100/1000 Base-TX Ethernet PCI-Express Adapter Device Driver (14104003) supports the 2-Port 10/100/1000 Base-TX PCI-Express Adapter (feature 5767).

The 4-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e414571614102004) supports the Gigabit Ethernet PCI-Express Adapter (feature 5899).

The Int Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203) Device Driver supports both the 1 Gigabit and 10 Gigabit Ethernet PCI- Express Adapter (feature 1769).

The Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203) Device Driver supports both the 1 Gigabit and 10 Gigabit Ethernet PCI- Express Adapter (feature 1768).

The Host Ethernet Adapter Device Driver supports the Host Ethernet Adapter (feature 5636, 5637, and 5639).

## Configuration Parameters

The following configuration parameter is supported by all Ethernet device drivers:

**Alternate Ethernet Addresses**
The device drivers support the device's hardware address as the network address or an alternate network address configured through software. When an alternate address is used, any valid Individual Address can be used. The least significant bit of an Individual Address must be set to zero. A multicast address cannot be defined as a network address. Two configuration parameters are provided to provide the alternate Ethernet address and enable the alternate address.

### *PCI Ethernet Device Driver (22100020)*

The PCI Ethernet Device Driver (22100020) supports the following additional configuration parameters:

**Full Duplex**
Indicates whether the adapter is operating in full-duplex or half-duplex mode. If this field is set to yes, the device driver programs the adapter to be in full-duplex mode.

**Hardware Transmit Queue**
Specifies the actual queue size the adapter uses to transmit packets. Each element corresponds to an Ethernet packet. It is configurable at 16, 32, 64, 1 28, and 256 elements.

**Hardware Receive Queue**
Specifies the actual queue size the adapter uses to receive packets. Each element corresponds to an Ethernet packet. It is configurable at 16, 32, 64, 128, and 256 elements.

### *10/100 Mbps Ethernet PCI Adapter Device Driver (23100020)*

The 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020) supports the following additional configuration parameters:

**Software Transmit Queue**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 16 through 16 384.

**Hardware Receive Queue**
The 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020) supports a user-configurable receive queue for the adapter. This is the actual queue the adapter uses to receive packets. Each element corresponds to an Ethernet packet. It is configurable at 16, 32, 64, 128, and 256 elements.

**Receive Buffer Pool**
The 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020) implements a private pool of receive memory buffers in order to enhance driver performance. The number of private receive buffers reserved by the driver is configurable from 16 to 2048 elements.

**Media Speed**
The 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020) supports a user-configurable media speed for the adapter. The media speed attribute indicates the speed at which the adapter attempts to operate. The available speeds are 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex and autonegotiation, with a default of autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select the specific speed.

**Note:** If autonegotiation is selected, the remote link device must also be set to autonegotiate or the link might not function properly.

**Inter Packet Gap**
The 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020) supports a user-configurable inter packet gap for the adapter. The inter packet gap attribute controls the aggressiveness of the adapter on the network. A small number increases the aggressiveness of the adapter, but a large number decreases the aggressiveness (and increase the fairness) of the adapter. A small number (more aggressive) could cause the adapter to capture the network by forcing other less aggressive nodes to defer. A larger number (less aggressive) might cause the adapter to defer more often than normal. If the statistics for other nodes on the network show a large number of collisions and deferrals, then try increasing this number. The default is 96, which results in IPG of 9.6 micro seconds for 10 Mbps and 0.96 microseconds for 100 Mbps media speed. Each unit of bit rate introduces an IPG of 100 nsec at 10 Mbps, and 10 nsec at 100 Mbps media speed.

**Link Polling Timer**
The 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020) implements a polling function (**Enable Link Polling**) that periodically queries the adapter to determine whether the Ethernet link is up or down. The **Enable Link Polling** attribute is disabled by default. If this function is enabled, the link polling timer value indicates how often the driver should poll the adapter for link status. This value can range from 100 to 1000 milliseconds. If the adapter's link goes down, the device driver disables its **NDD_RUNNING** flag. When the device driver finds that the link has come back up, it enables this **NDD_RUNNING** flag. In order for this to work successfully, protocol layer implementations, such as Etherchannel, need notification if the link has gone down. Enable the **Enable Link Polling** attribute to obtain this notification. Because of the additional PIO calls that the device driver makes, enabling this attribute can decrease the performance of this adapter.

### *10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01)*

The 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) supports the following additional configuration parameters:

**Software Transmit Queue**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Hardware Transmit Queue**
The 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) supports a user-configurable transmit queue for the adapter. This is the actual queue the adapter uses to transmit packets. Each element corresponds to an Ethernet packet. It is configurable from 100 to 1024 elements.

**Hardware Receive Queue**
The 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) supports a user-configurable receive queue for the adapter. This is the actual queue the adapter uses to receive packets. Each element corresponds to an Ethernet packet. It is configurable from 100 to 1024 elements.

**Receive Buffer Pool**
The 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) implements a private pool of receive memory buffers in order to enhance driver performance. The number of private receive buffers reserved by the driver is configurable from 512 to 2048 elements.

**Media Speed**
The 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) supports a user-configurable media speed for the adapter. The media speed attribute indicates the speed at which the adapter attempts to operate. The available speeds are 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex and autonegotiation, with a default of autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select the specific speed.

**Note:** If autonegotiation is selected, the remote link device must also be set to autonegotiate or the link might not function properly.

**Link Polling Timer**
The 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) implements a polling function which periodically queries the adapter to determine whether the Ethernet link is up or down. If this function is enabled, the link polling timer value indicates how often the driver should poll the adapter for link status. This value can range from 100 to 1000 milliseconds.

**Checksum Offload**
The 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) supports the capability of the adapter to calculate TCP checksums in hardware. If this capability is enabled, the TCP checksum calculation is performed on the adapter instead of the host, which can increase system performance. Possible values are Yes and No.

**Transmit TCP Resegmentation Offload**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the **Yes** and **No** values.

**IPsec Offload**
The 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) supports the capability of the adapter to perform IPsec cryptographic algorithms for data encryption and authentication in hardware. This capability enables the host to offload processor-intensive cryptographic processing to the adapter, which can increase system performance. Possible values are Yes and No.

### *Gigabit Ethernet-SX PCI Adapter Device Driver (14100401)*
The Gigabit Ethernet-SX PCI Adapter Device Driver (14100401) supports the following additional configuration parameters:

**Software Transmit Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Checksum Offload**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the checksum for the frame.

**Media Speed**
The Gigabit Ethernet-SX PCI Adapter Device Driver (14100401) supports a user-configurable media speed only for the IBM® 10/100/1000 Base-T Ethernet PCI adapter (feature 2975). For the Gigabit Ethernet-SX PCI Adapter (feature 2969), the only possible choice is autonegotiation. The media speed attribute indicates the speed at which the adapter attempts to operate. The available speeds are 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex and autonegotiation, with a default of autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select the specific speed.

**Note:** The autonegotiation setting must be selected in order for the adapter to run at 1000 M-bit/s.

**Enable Hardware Transmit TCP Resegmentation**
When you set the attribute value to **Yes**, the adapter performs TCP resegmentation on transmitted TCP segments. With this capability, TCP/IP can send larger datagrams to the adapter, which can increase performance. When you set the attribute value to **No**, TCP resegmentation is not performed.

**Note:** The default values for the Gigabit Ethernet-SX PCI Adapter Device Driver (14100401) configuration parameters were chosen for optimal performance. Do not change these default values.

The following configuration parameters for the Gigabit Ethernet-SX PCI Adapter Device Driver (14100401) are not accessible using SMIT. You can modify them only using the **chdev** command.

**stat_ticks**
Indicates the number of microseconds that the adapter waits before updating the adapter statistics (through a DMA write) and generating an interrupt to the host. Valid values range from 1000 through 1 000 000. The default value is 1 000 000.

**receive_ticks**
Indicates the number of microseconds that the adapter waits before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 1000, the default value is 50.

**receive_bds**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 128. The default value is 6.

**tx_done_ticks**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 1 000 000. The default value is 1 000 000.

**tx_done_count**
Indicates the number of transmit buffers that the adapter transfers from host memory before updating the send consumer index (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 128. The default value is 64.

**receive_proc**
When this number of receive buffer descriptors is processed by the device driver (or all packets are received), the device driver returns this number of receive buffer descriptors to the adapter through an MMIO write. Valid values range from 1 through 64. The default value is 16.

**rxdesc_count**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the `rx_handler()` routine and continues processing other adapter

events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions and adapter status changes) that are processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

### *Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802)*

The Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values range from 128 through 1024.

**Receive Descriptor Queue Size**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values range from 128 through 1024.

**Software Transmit Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Media Speed**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 Mbps full-duplex and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the duplexity. When the network does not support autonegotiation, select 1000 Mbps full-duplex.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Transmit TCP Resegmentation**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the values of **Yes** and **No**.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter performs TCP resegmentation for the frame.

**Enable Hardware Transmit and Receive Checksum**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the TCP checksum for the frame.

The following configuration parameters for the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802) are not accessible using SMIT. You can modify them only using the **chdev** command.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the **rx_handler()** routine and continues processing other adapter events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions, and adapter status changes) that are processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

**delay_open**
When you set the attribute value to **Yes**, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of **Yes** and **No**. The default value is **No**.

### *10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902)*
The 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values range from 128 through 1024.

**Receive Descriptor Queue Size**
Indicates the maximum number of received Ethernet packets that the adapter can buffer. Valid values range from 128 through 1024.

**Software Transmit Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Media Speed**
Indicates the speed at which the adapter attempts to operate. The available speeds are 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex, 1000 Mbps full-duplex, and autonegotiation, with a default of autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select the specific speed.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Transmit TCP Resegmentation**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the values of **Yes** and **No**.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter performs TCP resegmentation for the frame.

**Enable Hardware Transmit and Receive Checksum**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the TCP checksum for the frame.

The following configuration parameters for the 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902) are not accessible using SMIT. You can modify them only using the **chdev** command.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the **rx_handler()** routine and continues processing other adapter events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions, and adapter status changes) that are processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

**delay_open**
When you set the attribute value to **Yes**, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of **Yes** and **No**. The default value is **No**.

**compat_mode**
Setting this attribute to Yes forces the adapter to implement an early version of the IEEE 802.3z autonegotiation protocol. Use the yes value only if the adapter is unable to establish a link with your older Gigabit Ethernet-TX adapters or switches. Valid values are yes and No. The default value is No.

**Note:** If this option is enabled, the adapter cannot establish a link with newer Gigabit Ethernet-TX hardware. Enable this option only if you cannot establish a link using autonegotiation, but can force a link at a slower speed (for example, 100 full-duplex).

### *2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802)*
The 2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values range from 128 through 1024.

**Receive Descriptor Queue Size**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values range from 128 through 1024.

**Software Transmit Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Media Speed**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 Mbps full-duplex and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the duplexity. When the network does not support autonegotiation, select 1000 Mbps full-duplex.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Transmit TCP Resegmentation**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the values of **Yes** and **No**.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter performs TCP resegmentation for the frame.

**Enable Hardware Transmit and Receive Checksum**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the TCP checksum for the frame.

**Enable Failover Mode**
Indicates the requested failover configuration for the port. You can specify the values of **primary**, **backup**, and **disable**.

| Item | Description |
|---|---|
| **primary** | Indicates the port is to act as the primary port in a failover configuration for a 2-port gigabit adapter. |
| **backup** | Indicates the port is to act as the backup port in a failover configuration for a 2-port gigabit adapter. |
| **disable** | Indicates the port is not a member of a failover configuration. This is the default value for failover. |

The following configuration parameters for the 2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802) are not accessible using SMIT. You can modify them only using the **chdev** command.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the **rx_handler()** routine and continues processing other adapter events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions, and adapter status changes) that are processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of

the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

**delay_open**
When you set the attribute value to **Yes**, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of **Yes** and **No**. The default value is **No**.

**failback**
This attribute is used with the **Failover Mode** attribute. If the **Failover Mode** attribute is enabled, setting this attribute to the **Yes** value causes the adapter to automatically fail back to the primary port if the primary port recovers. You can specify the values of **Yes** and **No**. The default value is **Yes**.

**failback_delay**
This attribute is used with the **failback** attribute. If the **failback** attribute is enabled, the **failback_delay** attribute specifies the number of seconds that the adapter waits before failing back to the primary port, after the primary port recovers. This delay is useful for ensuring that the primary port has fully recovered and for allowing switch protocols (for example, Spanning Tree Protocol) to complete. Valid values range from 0 through 300 seconds. Setting the **failback_delay** attribute to 0 seconds disables the delay timer, causing failback to occur immediately. The default value is 15 seconds.

### *2-Port 10/100/1000 Base-TX PCI-X Adapter (14108902)*
The 2-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14108902) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values range from 128 through 1024.

**Receive Descriptor Queue Size**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values range from 128 through 1024.

**Software Transmit Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Media Speed**
Indicates the speed at which the adapter attempts to operate. The available speeds are 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex, 1000 Mbps full-duplex, and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select the specific speed.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Transmit TCP Resegmentation**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the values of **Yes** and **No**.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter performs TCP resegmentation for the frame.

**Enable Hardware Transmit and Receive Checksum**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the TCP checksum for the frame.

**Failover Mode (failover)**
Indicates the requested failover configuration for the port. You can specify the values of **primary**, **backup**, and **disable**.

| Item | Description |
|---|---|
| **primary** | Indicates the port is to act as the primary port in a failover configuration for a 2-port gigabit adapter. |
| **backup** | Indicates the port is to act as the backup port in a failover configuration for a 2-port gigabit adapter. |
| **disable** | Indicates the port is not a member of a failover configuration. This is the default value for failover. |

The following configuration parameters for the 2-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14108902) are not accessible using SMIT. You can modify them only using the **chdev** command.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the **rx_handler()** routine and continues processing other adapter events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions, and adapter status changes) that are processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

**delay_open**
When you set the attribute value to **Yes**, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of **Yes** and **No**. The default value is **No**.

**failback**
This attribute is used with the **Failover Mode** attribute. If the **Failover Mode** attribute is enabled, setting this attribute to the **Yes** value causes the adapter to automatically fail back to the primary port if the primary port recovers. You can specify the values of **Yes** and **No**. The default value is **Yes**.

**failback_delay**
This attribute is used with the **failback** attribute. If the **failback** attribute is enabled, the **failback_delay** attribute specifies the number of seconds that the adapter waits before failing back to the primary port, after the primary port recovers. This delay is useful for ensuring that the primary port has fully recovered and for allowing switch protocols (for example, Spanning Tree Protocol) to complete. Valid values range from 0 through 300 seconds. Setting the **failback_delay** attribute to 0 seconds disables the delay timer, causing failback to occur immediately. The default value is 15 seconds.

**compat_mode**
When you set the attribute value to**Yes**, the adapter is forced to implement an early version of the IEEE 802.3z autonegotiation protocol. Use the **Yes** value only if the adapter is unable to establish a link with your older Gigabit Ethernet-TX adapters or switches. You can specify the values of **Yes** and **No**. The default value is **No**.

**Note:** If this option is enabled, the adapter cannot establish a link with newer Gigabit Ethernet-TX hardware. Enable this option only if you cannot establish a link using autonegotiation, but can force a link at a slower speed (for example, 100 full-duplex).

### *10 Gigabit Ethernet-SR PCI-X Adapter (1410ba02) and 10 Gigabit Ethernet-LR PCI_X Adapter (1410bb02)*

The 10 Gigabit Ethernet-SR PCI-X Adapter (1410ba02) and 10 Gigabit Ethernet-LR PCI_X Adapter (1410bb02) support the following configuration parameters:

**Transmit Descriptor Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values range from 128 through 1024.

**Receive Descriptor Queue Size**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values range from 128 through 1024.

**Software Transmit Queue**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Transmit TCP Resegmentation Offload**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the **Yes** and **No** values.

**Enable Hardware Checksum Offload**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the checksum for the frame.

### *10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter (1410eb02) and 10 Gigabit Ethernet-LR PCI-X 2.0 DDR Adapter (1410ec02)*

The 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter (1410eb02) and the 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter (1410ec02) support the following configuration parameters:

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. The TCP/IP settings for the interface associated with the adapter are automatically initialized to maximum transmission unit MTU 9000 when the **Yes** value is selected. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes and the MTU is 1500. When the jumbo frame setting is enabled, it might be possible to communicate only with other network nodes that are also jumbo-enabled and that have the same MTU. This feature can result in a considerable performance improvement. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Transmit TCP Resegmentation**
Permits the adapter to perform resegmentation of transmitted TCP packets that are transmitted over IPv4 and IPv6 in hardware. With this capability, the host can create TCP segments that are larger than the actual MTU size of the Ethernet link (packet sizes of up to 64 KB can be created). The adapter then

subdivides these very large packets into multiple MTU-size packets. This offloading of packet creation is recommended for increased system performance. You can specify the values of **Yes** and **No**.

**Enable Hardware Transmit and Receive Checksum Offload**
When you set this attribute to **Yes**, the adapter calculates the checksum for TCP frames transmitted and received over IPv4 and IPv6. This setting is suggested for improved system performance. When you specify the **No** value, the checksum is calculated by the appropriate system software.

**Note:** The `mbuf` structure, which describes a transmitted frame, contains a flag that indicates whether the adapter must calculate the checksum for the frame.

**Enable Hardware Receive UDP Checksum Offload**
Setting this attribute to the Yes value indicates that the adapter calculates the checksum for UDP standard and fragmented frames received over IPv4 and IPv6. This setting is suggested for improved system performance. If you specify the No value, the checksum is calculated by the appropriate system software.

**Note:** The `mbuf` structure that describes a received frame contains a flag that indicates whether the adapter calculated the checksum for the frame.

### *Gigabit Ethernet-SX Adapter Device Driver (e414a816)*
The Gigabit Ethernet-SX Adapter Device Driver (e414a816) supports the following additional configuration parameters:

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Checksum Offload**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the checksum for the frame.

**Media Speed**
The Gigabit Ethernet-SX Adapter Device Driver (e414a816) supports a user-configurable media speed for 1000 Mbps full-duplex and autonegotiation. The media speed attribute indicates the speed at which the adapter attempts to operate. Select autonegotiate when the adapter should use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select the specific speed.

**Note:** The default values for the Gigabit Ethernet-SX Adapter Device Driver (e414a816) configuration parameters were chosen for optimal performance. Do not change these default values.

The following configuration parameters for the Gigabit Ethernet-SX Adapter Device Driver (e414a816) are not accessible using SMIT. You can modify them only using the **chdev** command.

**stat_ticks**
Indicates the number of microseconds that the adapter waits before updating the adapter statistics (through a DMA write) and generating an interrupt to the host. Valid values range from 1000 through 1 000 000. The default value is 1 000 000.

**receive_ticks**
Indicates the number of microseconds that the adapter waits before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 1000, the default value is 50.

**receive_bds**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 128. The default value is 6.

**tx_done_ticks**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 1 000 000. The default value is 1 000 000.

**tx_done_count**
Indicates the number of transmit buffers that the adapter transfers from host memory before updating the send consumer index (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 128. The default value is 64.

**receive_proc**
When this number of receive buffer descriptors is processed by the device driver (or all packets are received), the device driver returns this number of receive buffer descriptors to the adapter through an MMIO write. Valid values range from 1 through 64. The default value is 16.

**rxdesc_count**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the `rx_handler()` routine and continues processing other adapter events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions and adapter status changes) that are processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

### *Gigabit Ethernet-SX Adapter Device Driver (14101403)*
The Gigabit Ethernet-SX Adapter Device Driver (14101403) supports the following additional configuration parameters:

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Checksum Offload**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the checksum for the frame.

**Media Speed**
The Gigabit Ethernet-SX Adapter Device Driver (14101403) supports a user-configurable media speed for 1000 Mbps full-duplex and autonegotiation. The media speed attribute indicates the speed at which the adapter attempts to operate. Select autonegotiate when the adapter should use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select the specific speed.

**Note:** The default values for the Gigabit Ethernet-SX Adapter Device Driver (14101403) configuration parameters were chosen for optimal performance. Do not change these default values.

The following configuration parameters for the Gigabit Ethernet-SX Adapter Device Driver (14101403) SMIT. You can modify them only using the **chdev** command.

**receive_ticks**
Indicates the number of microseconds that the adapter waits before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 1000, the default value is 50.

**receive_bds**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 128. The default value is 6.

**tx_done_ticks**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 1 000 000. The default value is 1 000 000.

**tx_done_count**
Indicates the number of transmit buffers that the adapter transfers from host memory before updating the send consumer index (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 128. The default value is 64.

**receive_proc**
When this number of receive buffer descriptors is processed by the device driver (or all packets are received), the device driver returns this number of receive buffer descriptors to the adapter through an MMIO write. Valid values range from 1 through 64. The default value is 16.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the **rx_handler()** routine and continues processing other adapter events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions, and adapter status changes) that are processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

**delay_open**
When you set the attribute value to **Yes**, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of **Yes** and **No**. The default value is **No**.

### *Gigabit Ethernet-SX Adapter Device Driver (14106703)*

The Gigabit Ethernet-SX Adapter Device Driver (14106703) supports the following additional configuration parameters:

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Checksum Offload**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the checksum for the frame.

**Media Speed**
Indicates the speed at which the adapter attempts to operate. You can specify the values of 1000 Mbps full-duplex and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the duplexity. When the network does not support autonegotiation, select 1000 Mbps full-duplex.

**Note:** The default values for the Gigabit Ethernet-SX Adapter Device Driver (14106703) configuration parameters are chosen for optimal performance. Do not change these default values.

The following configuration parameters for the Gigabit Ethernet-SX Adapter Device Driver (14106703) are not accessible using SMIT. You can modify them only using the **chdev** command.

**receive_ticks**
Indicates the number of microseconds that the adapter waits before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 1000. The default value is 50.

**receive_bds**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 128. The default value is 6.

**tx_done_ticks**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 1 000 000. The default value is 1 000 000.

**tx_done_count**
Indicates the number of transmit buffers that the adapter transfers from host memory before updating the send consumer index (through a DMA write) and generating an interrupt to the host. Valid values range from 0 through 128. The default value is 64.

**receive_proc**
When this number of receive buffer descriptors is processed by the device driver (or all packets are received), the device driver returns this number of receive buffer descriptors to the adapter through an MMIO write. Valid values range from 1 through 64. The default value is 16.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the **rx_handler()** routine and continues processing other adapter events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions, and adapter status changes) that is processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

**delay_open**
When you set the attribute value to **Yes**, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands, such as the **ifconfig** command, however, might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of **Yes** and **No**. The default value is **No**.

### *4-Port 10/100/1000 Base-TX PCI-X Adapter (14101103)*

The 4-Port 10/100/1000 Base-TX PCI-X Adapter (14101103) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values range from 128 through 1024.

**Receive Descriptor Queue Size**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values range from 128 through 1024.

**Software Transmit Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Media Speed**
Indicates the speed at which the adapter attempts to operate. The available speeds are 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex, 1000 Mbps full-duplex, and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select the specific speed.

**Note:** 1000 Mbps half-duplex is not a valid value. The IEEE 802.3z specification dictates that the gigabit speeds for half-duplex must be autonegotiated for copper (TX)-based adapters. Select autonegotiation if this speed is required.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Transmit TCP Resegmentation Offload**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the **Yes** and **No** values.

**Enable Hardware Checksum Offload**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the checksum for the frame.

**Gigabit Backward Compatibility**
Older gigabit TX equipment might not be able to communicate with this adapter. If the adapter is unable to communicate with your older gigabit equipment, enabling this option forces the adapter to implement the IEEE 802.3z incorrectly. As such, this option should be enabled if the adapter is unable to communicate with your older gigabit equipment.

**Note:** Enabling this option forces the adapter to implement the IEEE 802.3z incorrectly. If this option is enabled, the adapter cannot communicate with newer equipment. Enable this option only if you cannot obtain a link using autonegotiation, but can force a link at a slower speed (for example, 100 full-duplex).

**Failover Mode (failover)**
Indicates the requested failover configuration for the port. You can specify the values of **primary**, **backup**, and **disable**. You can change this attribute using SMIT.

| Item | Description |
|---|---|
| **primary** | Indicates the port is to act as the primary port in a failover configuration for a 2-port gigabit adapter. |
| **backup** | Indicates the port is to act as the backup port in a failover configuration for a 2-port gigabit adapter. |
| **disable** | Indicates the port is not a member of a failover configuration. This is the default value for failover. |

### *4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)*
The 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values range from 128 through 1024.

**Receive Descriptor Queue Size**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values range from 128 through 1024.

**Software Transmit Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Media Speed**
Indicates the speed at which the adapter attempts to operate. The available speeds are 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex, 1000 Mbps full-duplex, and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter must use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select the specific speed.

**Note:** 1000 Mbps half-duplex is not a valid value. The IEEE 802.3z specification dictates that the gigabit speeds for half-duplex must be autonegotiated for copper (TX)-based adapters. Select autonegotiation if this speed is required.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Transmit TCP Resegmentation Offload**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the **Yes** and **No** values.

**Enable Hardware Checksum Offload**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the checksum for the frame.

**Gigabit Backward Compatibility**
Forces the adapter to implement the IEEE 802.3z incorrectly. Older gigabit TX equipment might not be able to communicate with the adapter. Enable the option if the adapter is unable to communicate with your older gigabit equipment.

**Important:** If the option is enabled, the adapter cannot communicate with newer equipment. Enable the option only if you cannot obtain a link using autonegotiation, but can force a link at a slower speed (for example, 100 full-duplex).

**Failover Mode (failover)**
Indicates the requested failover configuration for the port. You can specify the values of **primary**, **backup**, and **disable**. You can change this attribute using SMIT.

| Item | Description |
|---|---|
| **primary** | Indicates the port is to act as the primary port in a failover configuration for a 2-port gigabit adapter. |
| **backup** | Indicates the port is to act as the backup port in a failover configuration for a 2-port gigabit adapter. |
| **disable** | Indicates the port is not a member of a failover configuration. This is the default value for failover. |

### *2-Port Gigabit Ethernet-SX PCI-Express Adapter (14103f03)*
The 2-Port Gigabit Ethernet-SX PCI-Express Adapter (14103f03) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values range from 128 through 1024.

**Receive Descriptor Queue Size**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values range from 128 through 1024.

**Software Transmit Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Media Speed**
Indicates the speed at which the adapter attempts to operate. You can specify the values of 1000 Mbps full-duplex and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the duplexity. When the network does not support autonegotiation, select 1000 Mbps full-duplex.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Transmit TCP Resegmentation Offload**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the **Yes** and **No** values.

**Enable Hardware Checksum Offload**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted TCP frames and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the checksum for the frame.

**Failover Mode (failover)**
Indicates the requested failover configuration for the port. You can specify the values of **primary**, **backup**, and **disable**. You can change this attribute using SMIT.

| Item | Description |
|---|---|
| **primary** | Indicates the port is to act as the primary port in a failover configuration for a 2-port gigabit adapter. |
| **backup** | Indicates the port is to act as the backup port in a failover configuration for a 2-port gigabit adapter. |
| **disable** | Indicates the port is not a member of a failover configuration. This is the default value for failover. |

The following configuration parameters for the 2-Port Gigabit Ethernet-SX PCI-Express Adapter Device Driver (14103f03) are not accessible using SMIT. You can modify them only using the **chdev** command.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the **rx_handler()** routine and continues processing other adapter events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions, and adapter status changes) that are processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

**delay_open**
When you set the attribute value to **Yes**, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of **Yes** and **No**. The default value is **No**.

**failback**
The attribute is used with the **Failover Mode** attribute. When the **Failover Mode** attribute is enabled, setting this attribute value to **Yes** causes the adapter to automatically fail back to the primary port if the primary port recovers. You can specify the values of **Yes** and **No**. The default value is **Yes**.

**failback_delay**
The attribute is used with the **failback** attribute. When the **failback** attribute is enabled, the **failback_delay** attribute specifies the number of seconds that the adapter waits before failing back to the primary port, after the primary port recovers. This delay is useful for ensuring that the primary port has fully recovered and for allowing switch protocols (for example, Spanning Tree Protocol) to complete. Valid values range from 0 through 300 seconds. Setting the **failback_delay** attribute to 0 seconds disables the delay timer, causing failback to occur immediately. The default value is 15 seconds.

### *2-Port 10/100/1000 Base-TX PCI-Express Adapter (14104003)*

The 2-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14104003) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values range from 128 through 1024.

**Receive Descriptor Queue Size**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values range from 128 through 1024.

**Software Transmit Queue Size**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 16 384.

**Media Speed**
Indicates the speed at which the adapter attempts to operate. The available speeds are 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex, 1000 Mbps full-duplex, and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select a specific speed.

**Note:** 1000 Mbps half-duplex is not a valid value. The IEEE 802.3z specification dictates that the gigabit speeds for half-duplex must be autonegotiated for copper (TX)-based adapters. Select autonegotiation if this speed is required.

**Transmit Jumbo Frames**
When you set the attribute value to **Yes**, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the attribute value to **No**, the maximum size of frames transmitted is 1518 bytes. Frames up to 9018 bytes in length can always be received on this adapter.

**Enable Hardware Transmit TCP Resegmentation**
Permits the adapter to perform resegmentation of transmitted TCP segments in hardware. With this capability, the host can use TCP segments that are larger than the actual MTU size of the Ethernet link, which can increase system performance. You can specify the values of **Yes** and **No**.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter performs TCP resegmentation for the frame.

**Enable Hardware Transmit and Receive Checksum**
When you set the attribute value to **Yes**, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to **No**, the checksum is calculated by appropriate software.

**Note:** The **mbuf** structure, which describes a transmitted frame, contains a flag that indicates whether the adapter calculates the TCP checksum for the frame.

**Failover Mode (failover)**
Indicates the requested failover configuration for the port. You can specify the attribute values of **primary**, **backup**, and **disable**.

| Item | Description |
|---|---|
| **primary** | Indicates the port is to act as the primary port in a failover configuration for a 2-port gigabit adapter. |
| **backup** | Indicates the port is to act as the backup port in a failover configuration for a 2-port gigabit adapter. |
| **disable** | Indicates the port is not a member of a failover configuration. This is the default value for failover. |

The following configuration parameters for the 2-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14104003) are not accessible using SMIT. You can modify them only using the **chdev** command.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the **rx_handler()** routine and continues processing other adapter events, such as transmit completions and adapter status changes. Valid values range from 1 through 1 000 000. The default value is 1000.

**slih_hog**
Indicates the number of adapter events (such as receive completions, transmit completions, and adapter status changes) that is processed by the device driver per interrupt. Valid values range from 1 through 1 000 000. The default value is 10.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the **mbuf** data area into DMA memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the **mbuf** structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an **mbuf** chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). Valid values range from 64 through 2048. The default value is 2048.

**delay_open**
When you set the attribute value to **Yes**, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command), however, might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of **Yes** and **No**. The default value is **No**.

**failback**
This attribute is used with the **Failover Mode** attribute. If the **Failover Mode** attribute is enabled, setting this attribute to the **Yes** value causes the adapter to automatically fail back to the primary port if the primary port recovers. You can specify the values of **Yes** and **No**. The default value is **Yes**.

**failback_delay**
This attribute is used with the **failback** attribute. If the **failback** attribute is enabled, the **failback_delay** attribute specifies the number of seconds that the adapter waits before failing back to the primary port, after the primary port recovers. This delay is useful for ensuring that the primary port has fully recovered and for allowing switch protocols (for example, Spanning Tree Protocol) to complete. Valid values range from 0 through 300 seconds. Setting the **failback_delay** attribute to 0 seconds disables the delay timer, causing failback to occur immediately. The default value is 15 seconds.

**compat_mode**
When you set the attribute value to **Yes**, the adapter is forced to implement an early version of the IEEE 802.3z autonegotiation protocol. Use the **Yes** value only if the adapter is unable to establish a link with your older Gigabit Ethernet-TX adapters or switches. You can specify the values of **Yes** and **No**. The default value is **No**.

**Note:** If this option is enabled, the adapter cannot establish a link with newer Gigabit Ethernet-TX hardware. Enable this option only if you cannot establish a link using autonegotiation, but can force a link at a slower speed (for example, 100 full-duplex).

### *Host Ethernet Adapter Device Driver*
The Host Ethernet Adapter Device Driver supports the following additional configuration parameters:

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the **Alternate Ethernet Address** attribute. You can set the attribute value to **Yes** or **No**. The default value is **No**.

**Enable RX and TX Checksum Offload of TCP Segments (tx_cksum and rx_cksum)**
Specifies whether the adapter is to perform receive checksum calculation and transmit checksum calculation for TCP segments. You can set the attribute value to **Yes** or **No**. The default value is **Yes**.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the attribute value to **Yes** or **No**. The default value is **Yes**. The setting is only applicable to a system that is not managed by Hardware Management Console (HMC) or by Integrated Virtualization Manager (IVM).

**Jumbo Frames (jumbo_frames)**
Specifies whether to transmit and receive jumbo frames, which range from 1522 bytes through 9022 bytes in size. You can set the attribute value to **Yes** or **No**. The default value is **No**. To enable the attribute, authorization from HMC or IVM is required. Disablement of this attribute is not dependent on HMC or IVM.

**Receive Processing Count (rx_proc)**
Indicates the number of receive descriptor that is processed before posting a work-queue element. The default value is determined per performance tuning.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter is to perform transmit TCP resegmentation for TCP segments. You can set the attribute value to **Yes** or **No**. The default value is **Yes**.

**Transmit Immediate Data Copy Threshold (tx_immd_copy)**
Specifies the maximum size for copying data into the immediate data area inside the transmit work-queue element. This value is tunable for performance. Possible values range from 136 through 224. The default value is 224.

**Requested Media Speed (media speed)**
Indicates the speed at which the adapter attempts to operate. The available speeds are 10 Mbps full-duplex, 100 Mbps full-duplex, 1000 Mbps full-duplex, 10 000 Mbps full-duplex, and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter must use autonegotiation across the network to determine the speed. When the network does not support autonegotiation, select a specific speed.

**Note:** Half-duplex is not a valid value. This attribute is only applicable to a system that is not managed by HMC or by IVM.

**Enable Multicore Scaling (Multicore)**
Enables or disables driver multithreading. You can set the attribute value to **Yes** or **No**. The default value is **Yes**. The multithreading value is specified by the HMC *MCS* parameter.

**Use Transmit Interface Specific Buffers (tx_isb)**
Enables or disables transmit copy optimizations. You can set the attribute value to **Yes** or **No**. The default value is **Yes**.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the attribute value to **Yes** or **No**. The default value is **No**.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. You can set the attribute to a value ranging from 1 through 64. The default value is 16, which demonstrates the best balance of performance versus latency.

### *4-Port Gigabit Ethernet PCI-Express Adapter (e414571614102004)*
4-Port Gigabit Ethernet PCI-Express Adapter (e414571614102004) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Media Speed (media_speed)**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 or 100 or 10 Mbps full-duplex, 100 or 10 Mbps half-duplex, and autonegotiation. The default value is autonegotiation. Select autonegotiate when the adapter must use autonegotiation across the network to determine the duplexity.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the alternate Ethernet address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the value of the attribute to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the value of the attribute to No, the checksum is calculated by using appropriate software.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the value of the attribute to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter must perform transmit TCP resegmentation for TCP segments. You can set the value of the attribute to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the value of the attribute to Yes or No. The default value is Yes.

**slih_hog**
Indicates the number of adapter events (such as, receive completions, transmit completions and adapter status changes) that are processed by the device driver per interrupt. Valid values are in the range1 - 1000000. The default value is 10.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the rx_handler() routine and continues to process other adapter events, such as transmit completions and adapter status changes. The valid values are in the range 1 - 1000000. The default value is 1000.

**ipv6_offload**
Enable large send and checksum offload for IPv6. You can set the attribute value to Yes or No. The default value is Yes.

**Number of Transmit Queues (tx_queues)**
Indicates the number of transmit queues used for transmission. Valid values range from 1 - 4. The default value is 1.

**Number of Receive Queues (rx_queues)**
Indicates the number of receive queues used for reception. Valid values range from 1 - 4.The default value is 1.

**receive_proc**
When this number of receive buffer descriptors is processed by the device driver (or all packets are received), the device driver returns this number of receive buffer descriptors to the adapter through an MMIO write. Valid values range from 1 - 64. The default value is 16.

**rxbuf_pool_sz**
Indicates the size of receive buffer pool . Valid values range from 512 - 8192. The default value is 4096.

**tx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a Direct Memory Access (DMA) write) and generating an interrupt to the host. Valid values range from 0 - 1023. The default value is 1023.

**rx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the receive consumer index (through a DMA) and generating an interrupt to the host. Valid values range from 0 - 1000. The default value is 50.

**tx_bds_done**
Indicates the number of transmit buffers that the adapter transfers to host memory before updating the index of the transmit return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 - 255. The default value is 0.

**rx_bds_done**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 - 128. The default value is 5.

### *1 Gigabit Ethernet 4-port Mezzanine Adapter (e4145616e4140518) and 1GbE 4-port Mezzanine Adapter (e4145616e4140528)*

The 1 Gibabit Ethernet 4-port Mezzanine Adapter Device Driver (e4145616e4140518) and 1 Gigabit Ethernet 4-port Mezzanine Adapter Device Driver (e4145616e4140528) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Media Speed (media_speed)**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 or 100 or 10 Mbps full-duplex, 100 or 10 Mbps half-duplex, and autonegotiation. The default value is

autonegotiation. Select autonegotiate when the adapter must use autonegotiation across the network to determine the duplexity.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the alternate Ethernet address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the value of the attribute to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the value of the attribute to No, the checksum is calculated by using appropriate software.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the value of the attribute to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter must perform transmit TCP resegmentation for TCP segments. You can set the value of the attribute to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the value of the attribute to Yes or No. The default value is Yes.

**slih_hog**
Indicates the number of adapter events (such as, receive completions, transmit completions and adapter status changes) that are processed by the device driver per interrupt. Valid values are in the range1 - 1000000. The default value is 10.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the rx_handler() routine and continues to process other adapter events, such as transmit completions and adapter status changes. The valid values are in the range 1 - 1000000. The default value is 1000.

**ipv6_offload**
Enable large send and checksum offload for IPv6. You can set the attribute value to Yes or No. The default value is Yes.

**Number of Transmit Queues (tx_queues)**
Indicates the number of transmit queues used for transmission. Valid values range from 1 - 4. The default value is 1.

**Number of Receive Queues (rx_queues)**
Indicates the number of receive queues used for reception. Valid values range from 1 - 4.The default value is 1.

**receive_proc**
When this number of receive buffer descriptors is processed by the device driver (or all packets are received), the device driver returns this number of receive buffer descriptors to the adapter through an MMIO write. Valid values range from 1 - 64. The default value is 16.

**rxbuf_pool_sz**
Indicates the size of receive buffer pool . Valid values range from 512 - 8192. The default value is 4096.

**tx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a Direct Memory Access (DMA) write) and generating an interrupt to the host. Valid values range from 0 - 1023. The default value is 1023.

**rx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the receive consumer index (through a DMA) and generating an interrupt to the host. Valid values range from 0 - 1000. The default value is 50.

**tx_bds_done**
Indicates the number of transmit buffers that the adapter transfers to host memory before updating the index of the transmit return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 - 255. The default value is 0.

**rx_bds_done**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values range from 0 - 128. The default value is 5.

### *4-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4143009)*
The 4-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a16e4143009) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Media Speed (media_speed)**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 or 100 or 10 Mbps full-duplex, 100 or 10 Mbps half-duplex, and autonegotiation. The default value is autonegotiation. Select autonegotiate when the adapter must use autonegotiation across the network to determine the duplexity.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the alternate Ethernet address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the value of the attribute to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the value of the attribute to No, the checksum is calculated by using appropriate software.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the value of the attribute to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter must perform transmit TCP resegmentation for TCP segments. You can set the value of the attribute to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the value of the attribute to Yes or No. The default value is Yes.

**slih_hog**
Indicates the number of adapter events (such as, receive completions, transmit completions and adapter status changes) that are processed by the device driver per interrupt. Valid values are in the range1 - 1000000. The default value is 10.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the rx_handler() routine and continues to process other adapter events, such as transmit completions and adapter status changes. The valid values are in the range 1 - 1000000. The default value is 1000.

**rxbuf_pool_sz**
Indicates the size of receive buffer pool . The valid values are in the range 256 - 2048. The default value is 1024.

**tx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a Direct Memory Access (DMA) write) and generating an interrupt to the host. Valid values are in the range 0 - 1023. The default value is 1023.

**rx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the receive consumer index (through a DMA) and generating an interrupt to the host. Valid values are in the range 0 - 1000. The default value is 50.

**tx_bds_done**
Indicates the number of transmit buffers that the adapter transfers to host memory before updating the index of the transmit return ring (through a DMA write) and generating an interrupt to the host. Valid values are in the range 0 - 255. The default value is 0.

**rx_bds_done**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values are in the range 0 - 128. The default value is 5.

### *2-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4140909)*

The 2-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a16e4140909) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Media Speed (media_speed)**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 or 100 or 10 Mbps full-duplex, 100 or 10 Mbps half-duplex, and autonegotiation. The default value is autonegotiation. Select autonegotiate when the adapter must use autonegotiation across the network to determine the duplexity.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the alternate Ethernet address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the value of the attribute to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the value of the attribute to No, the checksum is calculated by using appropriate software.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the value of the attribute to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter must perform transmit TCP resegmentation for TCP segments. You can set the value of the attribute to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the value of the attribute to Yes or No. The default value is Yes.

**slih_hog**
Indicates the number of adapter events (such as, receive completions, transmit completions and adapter status changes) that are processed by the device driver per interrupt. Valid values are in the range1 - 1000000. The default value is 10.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the rx_handler() routine and continues to process other adapter events, such as transmit completions and adapter status changes. The valid values are in the range 1 - 1000000. The default value is 1000.

**rxbuf_pool_sz**
Indicates the size of receive buffer pool . The valid values are in the range 256 - 2048. The default value is 1024.

**tx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a Direct Memory Access (DMA) write) and generating an interrupt to the host. Valid values are in the range 0 - 1023. The default value is 1023.

**rx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the receive consumer index (through a DMA) and generating an interrupt to the host. Valid values are in the range 0 - 1000. The default value is 50.

**tx_bds_done**
Indicates the number of transmit buffers that the adapter transfers to host memory before updating the index of the transmit return ring (through a DMA write) and generating an interrupt to the host. Valid values are in the range 0 - 255. The default value is 0.

**rx_bds_done**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values are in the range 0 - 128. The default value is 5.

### *2-Port Gigabit Ethernet PCI-Express Combo Adapter (e4143a161410a003)*
The 2-Port Gigabit Ethernet PCI-Express Combo Adapter device driver (e4143a161410a003) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Media Speed (media_speed)**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 or 100 or 10 Mbps full-duplex, 100 or 10 Mbps half-duplex, and autonegotiation. The default value is autonegotiation. Select autonegotiate when the adapter must use autonegotiation across the network to determine the duplexity.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the alternate Ethernet address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the value of the attribute to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the value of the attribute to No, the checksum is calculated by using appropriate software.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the value of the attribute to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter must perform transmit TCP resegmentation for TCP segments. You can set the value of the attribute to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the value of the attribute to Yes or No. The default value is Yes.

**slih_hog**
Indicates the number of adapter events (such as, receive completions, transmit completions and adapter status changes) that are processed by the device driver per interrupt. Valid values are in the range1 - 1000000. The default value is 10.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the rx_handler() routine and continues to process other adapter events, such as transmit completions and adapter status changes. The valid values are in the range 1 - 1000000. The default value is 1000.

**rxbuf_pool_sz**
Indicates the size of receive buffer pool . The valid values are in the range 256 - 2048. The default value is 1024.

**tx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a Direct Memory Access (DMA) write) and generating an interrupt to the host. Valid values are in the range 0 - 1023. The default value is 1023.

**rx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the receive consumer index (through a DMA) and generating an interrupt to the host. Valid values are in the range 0 - 1000. The default value is 50.

**tx_bds_done**
Indicates the number of transmit buffers that the adapter transfers to host memory before updating the index of the transmit return ring (through a DMA write) and generating an interrupt to the host. Valid values are in the range 0 - 255. The default value is 0.

**rx_bds_done**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values are in the range 0 - 128. The default value is 5.

### *2-Port Integrated Gigabit Ethernet PCI-Express Adapter (e4143a161410ed03)*

The 2-Port Integrated Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a161410ed03) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Media Speed (media_speed)**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 or 100 or 10 Mbps full-duplex, 100 or 10 Mbps half-duplex, and autonegotiation. The default value is autonegotiation. Select autonegotiate when the adapter must use autonegotiation across the network to determine the duplexity.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the alternate Ethernet address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the value of the attribute to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the value of the attribute to No, the checksum is calculated by using appropriate software.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the value of the attribute to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter must perform transmit TCP resegmentation for TCP segments. You can set the value of the attribute to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the value of the attribute to Yes or No. The default value is Yes.

**slih_hog**
Indicates the number of adapter events (such as, receive completions, transmit completions and adapter status changes) that are processed by the device driver per interrupt. Valid values are in the range1 - 1000000. The default value is 10.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**rx_hog**
When this number of receive buffer descriptors is processed by the device driver (or all packets were received), the device driver exits the rx_handler() routine and continues to process other adapter events, such as transmit completions and adapter status changes. The valid values are in the range 1 - 1000000. The default value is 1000.

**rxbuf_pool_sz**
Indicates the size of receive buffer pool . The valid values are in the range 256 - 2048. The default value is 1024.

**tx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the send consumer index (through a Direct Memory Access (DMA) write) and generating an interrupt to the host. Valid values are in the range 0 - 1023. The default value is 1023.

**rx_ticks_done**
Indicates the number of microseconds that the adapter waits before updating the receive consumer index (through a DMA) and generating an interrupt to the host. Valid values are in the range 0 - 1000. The default value is 50.

**tx_bds_done**
Indicates the number of transmit buffers that the adapter transfers to host memory before updating the index of the transmit return ring (through a DMA write) and generating an interrupt to the host. Valid values are in the range 0 - 255. The default value is 0.

**rx_bds_done**
Indicates the number of receive buffers that the adapter transfers to host memory before updating the producer index of the receive return ring (through a DMA write) and generating an interrupt to the host. Valid values are in the range 0 - 128. The default value is 5.

### *10 Gigabit Ethernet PCI Express Dual Port Adapter (7710008077108001)*
The 10 Gb Ethernet PCI Express Dual Port Adapter Device Driver (7710008077108001) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the alternate Ethernet address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the value of the attribute to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the value of the attribute to No, the checksum is calculated by using appropriate software.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the value of the attribute to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter must perform transmit TCP resegmentation for TCP segments. You can set the value of the attribute to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the value of the attribute to Yes or No. The default value is Yes.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**rx_intr_delay**
Interrupt delay timer sets an overall amount of time to delay an interrupt. This timer starts when the first packet arrives. When the timer expires, an interrupt is generated. This value is in microseconds. The valid values are in the range 0 - 65535 . The default value is 100.

**rx_ipkt_idelay**
Inter-packet interrupt delay timer sets an amount of time to hold off interrupts between packets. This timer starts when a packet arrives. The timer is reset anytime another packet arrives before the timer expires. If a packet does not arrive before the timer expires, an interrupt is generated. This value is in microseconds. The valid values are in the range 0 - 65535. The default value is 4.

**rx_intr_limit**
Indicates maximum packets processed by RX handler per interrupt. The valid values are in the range 1000 - 1000000. The default value is 1000.

### *10 Gigabit Ethernet-SR PCI Express Dual Port Adapter (771000801410b003)*

The 10 Gigabit Ethernet-SR PCI Express Dual Port Adapter Device Driver (771000801410b003) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the alternate Ethernet address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the value of the attribute to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the value of the attribute to No, the checksum is calculated by using appropriate software.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the value of the attribute to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter must perform transmit TCP resegmentation for TCP segments. You can set the value of the attribute to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the value of the attribute to Yes or No. The default value is Yes.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**rx_intr_delay**
Interrupt delay timer sets an overall amount of time to delay an interrupt. This timer starts when the first packet arrives. When the timer expires, an interrupt is generated. This value is in microseconds. The valid values are in the range 0 - 65535 . The default value is 100.

**rx_ipkt_idelay**
Inter-packet interrupt delay timer sets an amount of time to hold off interrupts between packets. This timer starts when a packet arrives. The timer is reset anytime another packet arrives before the timer expires. If a packet does not arrive before the timer expires, an interrupt is generated. This value is in microseconds. The valid values are in the range 0 - 65535. The default value is 4.

**rx_intr_limit**
Indicates maximum packets processed by RX handler per interrupt. The valid values are in the range 1000 - 1000000. The default value is 1000.

### *2-port 10GbE SR PCIe2 Adapter (a21910071410d003)*

The 2-port 10GbE SR PCIe2 Adapter Device Driver (a21910071410d003) supports the 10 Gigabit Ethernet PCI-Express Adapter.

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the mbuf data area into Direct Memory Access (DMA) memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the mbuf structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an mbuf chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). The valid values are in the range 64 - 64000. The default value is 2048.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 1, 2, 4, 8, 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the attribute value to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter is to perform transmit TCP resegmentation for TCP segments. You can set the attribute value to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the attribute value to Yes or No. The default value is Yes.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the Alternate Ethernet Address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**ipv6_offload**
Enable large send and checksum offload for IPv6. You can set the value of the attribute to Yes or No. The default value is Yes.

**intr_rate**
This is the maximum number of RX interrupts per second generated by the adapter. The valid values are in the range 0 - 65535. The default value is 20000.

**Number of Transmit Queues (tx_queues)**
Indicates the number of transmit queues used for transmission. The valid values are in the range 10 - 5. The default value is 2.

**Number of Receive Queues (rx_queues)**
Indicates the number of receive queues used for reception. The valid values are in the range 1 - 5. The default value is 3.

**receive limit (rx_limit)**
Indicates the maximum number of receive packets processed per interrupt. The valid values are in the range 1000 - 100000. The default value is 1000.

**rx_dynamic**
Indicates a receive side performance tunable, these are preferred values for POWER5, POWER6®, POWER7® or later. For POWER5 and POWER6 the value should be NO and for POWER7 or later the value should be YES. By default, the value is set to Yes.

### *PCIe2 2-port 10GbE SFP+Copper Adapter (a21910071410d103)*
The PCIe2 2-port 10GbE SFP+Copper Adapter Device Driver (a21910071410d103) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the attribute value to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to No, the checksum is calculated by appropriate software.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the mbuf data area into Direct Memory Access (DMA) memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the mbuf structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an mbuf chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). The valid values are in the range 64 - 64000. The default value is 2048.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 1, 2, 4, 8, 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the attribute value to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter is to perform transmit TCP resegmentation for TCP segments. You can set the attribute value to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the attribute value to Yes or No. The default value is Yes.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the Alternate Ethernet Address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**ipv6_offload**
Enable large send and checksum offload for IPv6. You can set the value of the attribute to Yes or No. The default value is Yes.

**intr_rate**
This is the maximum number of RX interrupts per second generated by the adapter. The valid values are in the range 0 - 65535. The default value is 20000.

**Number of Transmit Queues (tx_queues)**
Indicates the number of transmit queues used for transmission. The valid values are in the range 10 - 5. The default value is 2.

**Number of Receive Queues (rx_queues)**
Indicates the number of receive queues used for reception. The valid values are in the range 1 - 5. The default value is 3.

**receive limit (rx_limit)**
Indicates the maximum number of receive packets processed per interrupt. The valid values are in the range 1000 - 100000. The default value is 1000.

**rx_dynamic**
Indicates a receive side performance tunable, these are preferred values for POWER5, POWER6, POWER7 or later. For POWER5 and POWER6 the value should be NO and for POWER7 or later the value should be YES. By default, the value is set to Yes.

### *10GbE 4-port Mezzanine PCIe Adapter (a2191007df1033e7)*
The 10GbE 4-port Mezzanine PCIe Adapter Device Driver (a2191007df1033e7) supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the attribute value to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to No, the checksum is calculated by appropriate software.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the mbuf data area into Direct Memory Access (DMA) memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the mbuf structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an mbuf chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). The valid values are in the range 64 - 64000. The default value is 2048.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 1, 2, 4, 8, 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the attribute value to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter is to perform transmit TCP resegmentation for TCP segments. You can set the attribute value to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the attribute value to Yes or No. The default value is Yes.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the Alternate Ethernet Address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**ipv6_offload**
Enable large send and checksum offload for IPv6. You can set the value of the attribute to Yes or No. The default value is Yes.

**intr_rate**
This is the maximum number of RX interrupts per second generated by the adapter. The valid values are in the range 0 - 65535. The default value is 20000.

**Number of Transmit Queues (tx_queues)**
Indicates the number of transmit queues used for transmission. The valid values are in the range 10 - 5. The default value is 2.

**Number of Receive Queues (rx_queues)**
Indicates the number of receive queues used for reception. The valid values are in the range 1 - 5. The default value is 3.

**receive limit (rx_limit)**
Indicates the maximum number of receive packets processed per interrupt. The valid values are in the range 1000 - 100000. The default value is 1000.

**rx_dynamic**
Indicates a receive side performance tunable, these are preferred values for POWER5, POWER6, POWER7 or later. For POWER5 and POWER6 the value should be NO and for POWER7 or later the value should be YES. By default, the value is set to Yes.

### *Int Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203)*

The Int Multifunction Card SR Optical 10GbE (a219100714100904) and the Base-TX 10/100/1000 1GbE (a21910071410d203) Device Driver supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the attribute value to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to No, the checksum is calculated by appropriate software.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the mbuf data area into Direct Memory Access (DMA) memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the mbuf structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an mbuf chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). The valid values are in the range 64 - 64000. The default value is 2048.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 1, 2, 4, 8, 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the attribute value to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter is to perform transmit TCP resegmentation for TCP segments. You can set the attribute value to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the attribute value to Yes or No. The default value is Yes.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the Alternate Ethernet Address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**ipv6_offload**
Enable large send and checksum offload for IPv6. You can set the value of the attribute to Yes or No. The default value is Yes.

**intr_rate**
This is the maximum number of RX interrupts per second generated by the adapter. The valid values are in the range 0 - 65535. The default value is 20000.

**Number of Transmit Queues (tx_queues)**
Indicates the number of transmit queues used for transmission. The valid values are in the range 10 - 5. The default value is 2.

**Number of Receive Queues (rx_queues)**
Indicates the number of receive queues used for reception. The valid values are in the range 1 - 5. The default value is 3.

**receive limit (rx_limit)**
Indicates the maximum number of receive packets processed per interrupt. The valid values are in the range 1000 - 100000. The default value is 1000.

**rx_dynamic**
Indicates a receive side performance tunable, these are preferred values for POWER5, POWER6, POWER7 or later. For POWER5 and POWER6 the value should be NO and for POWER7 or later the value should be YES. By default, the value is set to Yes.

**Media Speed (media_speed)**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 or 100 or 10 Mbps full-duplex, and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the duplexity.

## *Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203)*

The Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203) Device Driver supports the following additional configuration parameters:

**Transmit Descriptor Queue Size (txdesc_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the adapter. Valid values are in the range 256 - 512. The default value is 512.

**Receive Descriptor Queue Size (rxdesc_que_sz)**
Indicates the maximum number of received Ethernet packets that the adapter can hold in its buffer. Valid values are 256, 512, 1024, and 2048. The default value is 2048.

**Software Transmit Queue Size (tx_que_sz)**
Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values are in the range 512 - 16384. The default value is 8192.

**Jumbo Frames (jumbo_frames)**
When you set the value of the attribute to Yes, frames up to 9018 bytes in length can be transmitted on this adapter. When you set the value of the attribute to No, the maximum size of frames transmitted is 1518 bytes.

**Enable Hardware Checksum Offload (chksum_offload)**
When you set the attribute value to Yes, the adapter calculates the checksum for transmitted and received TCP frames. When you set the attribute value to No, the checksum is calculated by appropriate software.

**delay_open**
When you set the value of the attribute to Yes, the adapter device driver delays its open completion until the Ethernet link status is determined to be either up or down. This prevents applications from sending data before the Ethernet link is established. Commands (for example, the **ifconfig** command) however might take longer to complete, especially when an active Ethernet link is not present. You can specify the values of Yes and No. The default value is No.

**copy_bytes**
When the number of data bytes in a transmit **mbuf** structure exceeds this value, the device driver maps the mbuf data area into Direct Memory Access (DMA) memory and updates the transmit descriptor so that it points to this DMA memory area. When the number of data bytes in a transmit **mbuf** structure does not exceed this value, the data is copied from the mbuf structure into a preallocated transmit buffer that is already mapped into DMA memory. The device driver also attempts to coalesce transmit data in an mbuf chain into a single preallocated transmit buffer until the total transmit data size exceeds that of the preallocated buffer (2048 bytes). The valid values are in the range 64 - 64000. The default value is 2048.

**Receive Packet Coalescing (rx_coalesce)**
Allows tuning of the RX packets that are coalesced before passing to a stack. Increasing this value impacts latency under certain scenarios. Decreasing this value might enhance large reception under limited scenarios. The valid values are 1, 2, 4, 8, 16, 32, 64, and 128. The default value is 16, which demonstrates the best balance of performance versus latency.

**Enable Receive TCP Segment Aggregation (large_receive)**
Enables coalescing receive packets into a larger packet before passing to the upper layer for enhanced performance. You can set the attribute value to Yes or No. The default value is Yes.

**Note:** Disable this value when the adapter is used in IP forwarding.

**Enable TX TCP Resegmentation Offload of TCP Segments (large_send)**
Specifies whether the adapter is to perform transmit TCP resegmentation for TCP segments. You can set the attribute value to Yes or No. The default value is Yes.

**Flow Control (flow_ctrl)**
Specifies whether the adapter enables transmit flow control and receive flow control. You can set the attribute value to Yes or No. The default value is Yes.

**Enable Alternate Ethernet Address (use_alt_addr)**
Specifies whether the adapter uses the MAC address that is specified in the Alternate Ethernet Address attribute. You can set the value of the attribute to Yes or No. The default value is No.

**ipv6_offload**
Enable large send and checksum offload for IPv6. You can set the value of the attribute to Yes or No. The default value is Yes.

**intr_rate**
This is the maximum number of RX interrupts per second generated by the adapter. The valid values are in the range 0 - 65535. The default value is 20000.

**Number of Transmit Queues (tx_queues)**
Indicates the number of transmit queues used for transmission. The valid values are in the range 10 - 5. The default value is 2.

**Number of Receive Queues (rx_queues)**
Indicates the number of receive queues used for reception. The valid values are in the range 1 - 5. The default value is 3.

**receive limit (rx_limit)**
Indicates the maximum number of receive packets processed per interrupt. The valid values are in the range 1000 - 100000. The default value is 1000.

**rx_dynamic**
Indicates a receive side performance tunable, these are preferred values for POWER5, POWER6, POWER7 or later. For POWER5 and POWER6 the value should be NO and for POWER7 or later the value should be YES. By default, the value is set to Yes.

**Media Speed (media_speed)**
Indicates the speed at which the adapter attempts to operate. The available speeds are 1000 or 100 or 10 Mbps full-duplex, and autonegotiation. The default is autonegotiation. Select autonegotiate when the adapter should use autonegotiation across the network to determine the duplexity.

## Interface Entry Points

For each Ethernet device, the interface to the device driver is achieved by calling the entry points for opening, closing, transmitting data, and issuing device control commands.

The following are interface entry points for an Ethernet device:

### *Device Driver Configuration and Unconfiguration*

The configuration entry points of the device drivers conform to the guidelines for kernel object file entry points.

These configuration entry points are as follows:

- `bent_config` for the Gigabit Ethernet-SX Adapter Device Driver (e414a816).
- `elxent_config` for the 2-port 10GbE SR PCIe2 Adapter (a21910071410d003).
- `elxent_config` for the PCIe2 2-port 10GbE SFP+Copper Adapter (a21910071410d103).
- `elxent_config` for the 10GbE 4-port Mezzanine PCIe Adapter (a2191007df1033e7).
- `elxent_config` for theInt Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203).
- `elxent_config` for the Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203).
- **gxent_config** for the Gigabit Ethernet-SX PCI Adapter Device Driver (14100401)
- **goent_config** for the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802), the 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902), the 2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802), the 2-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14108902), the 4-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14101103), and 4-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14106803), the 2-Port Gigabit Ethernet-SX PCI-Express Adapter Device Driver (14103f03), and the 2-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14104003).
- `hea_config` for Host Ethernet Adapter Device Driver.
- **kent_config** for the PCI Ethernet Device Driver (22100020)
- `kngent_config` for the 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter Device Driver (1410eb02) and the 10 Gigabit Ethernet-LR PCI_X 2.0 DDR Adapter Device Driver (1410ec02).
- **phxent_config** for the 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020)
- `ment_config` for the Gigabit Ethernet-SX Adapter Device Driver (14101403) and the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106703).
- `msnent_config` for the 10 Gb Ethernet PCI Express Dual Port Adapter (7710008077108001).
- `msnent_config` for the 10 Gb Ethernet-SR PCI Express Dual Port Adapter (771000801410b003).

- musent_config for the 4-Port Gigabit Ethernet PCI-Express Adapter (e414571614102004).
- musent_config for the 1GbE 4-port Mezzanine Adapter (e4145616e4140518) and 1GbE 4-port Mezzanine Adapter (e4145616e4140528).
- nment_config for the 4-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4143009).
- nment_config for the 2-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4140909).
- nment_config for the 2-Port Gigabit Ethernet PCI-Express Combo Adapter (e4143a161410a003).
- nment_config for the 2-Port Integrated Gigabit Ethernet PCI-Express Adapter (e4143a161410ed03).
- **scent_config** for the 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01)
- vent_config for the 10 Gigabit Ethernet-SR PCI-X Adapter Device Driver (1410ba02) and the 10 Gigabit Ethernet-LR PCI_X Adapter Device Driver (1410bb02).

### *Device Driver Open*

The open entry point for the device drivers perform a synchronous open of the specified network device.

The device driver issues commands to start the initialization of the device. The state of the device now is OPEN_PENDING. The device driver invokes the open process for the device. The open process involves a sequence of events that are necessary to initialize and configure the device. The device driver does the sequence of events in an orderly fashion to make sure that one step is finished executing on the adapter before the next step is continued. Any error during these sequence of events makes the open fail. The device driver requires about 2 seconds to open the device. When the whole sequence of events is done, the device driver verifies the open status and then returns to the caller of the open with a return code to indicate open success or open failure.

After the device has been successfully configured and connected to the network, the device driver sets the device state to **OPENED**, the **NDD_RUNNING** flag in the **NDD** flags field is turned on. In the case of unsuccessful open, both the **NDD_UP** and **NDD_RUNNING** flags in the **NDD** flags field are off and a non-zero error code is returned to the caller.

The open entry points are as follows:

- bent_open for the Gigabit Ethernet-SX Adapter Device Driver (e414a816).
- elxent_open for the 2-port 10GbE SR PCIe2 Adapter (a21910071410d003).
- elxent_open for the PCIe2 2-port 10GbE SFP+Copper Adapter (a21910071410d103).
- elxent_open for the 10GbE 4-port Mezzanine PCIe Adapter (a2191007df1033e7) .
- elxent_open for the Int Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203).
- elxent_open for the Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203).
- **gxent_open** for the Gigabit Ethernet-SX PCI Adapter Device Driver (14100401).
- **goent_open** for the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802), the 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902), the 2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802), the 2-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14108902), the 4-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14101103), and 4-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14106803), the 2-Port Gigabit Ethernet-SX PCI-Express Adapter Device Driver (14103f03), and the 2-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14104003).
- hea_open for the Host Ethernet Adapter Device Driver.
- **kent_open** for the PCI Ethernet Device Driver (22100020).
- kngent_open for the 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter Device Driver (1410eb02) and the 10 Gigabit Ethernet-LR PCI_X 2.0 DDR Adapter Device Driver (1410ec02).
- ment_open for the Gigabit Ethernet-SX Adapter Device Driver (14101403) and the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106703).

- `msnent_open` for the 10 Gb Ethernet PCI Express Dual Port Adapter (7710008077108001).
- `msnent_open` for the 10 Gb Ethernet-SR PCI Express Dual Port Adapter (771000801410b003).
- `musent_open` for the 4-Port Gigabit Ethernet PCI-Express Adapter (e414571614102004).
- `musent_open` for the 1GbE 4-port Mezzanine Adapter (e4145616e4140518) and 1GbE 4-port Mezzanine Adapter (e4145616e4140528).
- `nment_open` for the 4-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4143009).
- `nment_open` for the 2-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4140909).
- `nment_open` for the 2-Port Gigabit Ethernet PCI-Express Combo Adapter (e4143a161410a003).
- `nment_open` for the 2-Port Integrated Gigabit Ethernet PCI-Express Adapter (e4143a161410ed03).
- **phxent_open** for the 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020).
- **scent_open** for the 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01).
- `vent_open` for the 10 Gigabit Ethernet-SR PCI-X Adapter Device Driver (1410ba02) and the 10 Gigabit Ethernet-LR PCI_X Adapter Device Driver (1410bb02).

### *Device Driver Close*

The close entry point for the device drivers is called to close the specified network device. This function resets the device to a known state and frees system resources associated with the device.

The device will not be detached from the network until the device's transmit queue drains. That is, the close entry point will not return until all packets have been transmitted or timed out. If the device is inoperable at the time of the close, the device's transmit queue does not have to drain.

At the beginning of the close entry point, the device state is set to be **CLOSE_PENDING**. The **NDD_RUNNING** flag in the **ndd_flags** is turned off. After the outstanding transmit queue is all done, the device driver starts a sequence of operations to deactivate the adapter and to free up resources. Before the close entry point returns to the caller, the device state is set to **CLOSED**.

The close entry points are as follows:

- `bent_close` for the Gigabit Ethernet-SX Adapter Device Driver (e414a816).
- `elxent_close` for the 2-port 10GbE SR PCIe2 Adapter (a21910071410d003).
- `elxent_close` for the PCIe2 2-port 10GbE SFP+Copper Adapter (a21910071410d103).
- `elxent_close` for the 10GbE 4-port Mezzanine PCIe Adapter (a2191007df1033e7).
- `elxent_close` for the Int Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203).
- `elxent_close` for the Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203).
- **gxent_close** for the Gigabit Ethernet-SX PCI Adapter Device Driver (14100401)
- **goent_close** for the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802), the 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902), the 2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802), the 2-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14108902), the 4-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14101103), and 4-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14106803), the 2-Port Gigabit Ethernet-SX PCI-Express Adapter Device Driver (14103f03), and the 2-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14104003).
- `hea_close` for the Host Ethernet Adapter Device Driver.
- **kent_close** for the PCI Ethernet Device Driver (22100020).
- `kngent_close` for the 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter Device Driver (1410eb02) and the 10 Gigabit Ethernet-LR PCI_X 2.0 DDR Adapter Device Driver (1410ec02).
- `ment_close` for the Gigabit Ethernet-SX Adapter Device Driver (14101403) and the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106703).
- `msnent_close` for the 10 Gb Ethernet PCI Express Dual Port Adapter (7710008077108001).

- `msnent_close` for the 10 Gb Ethernet-SR PCI Express Dual Port Adapter (771000801410b003).
- `musent_close` for the 4-Port Gigabit Ethernet PCI-Express Adapter (e414571614102004).
- `musent_close` for the 1GbE 4-port Mezzanine Adapter (e4145616e4140518) and 1GbE 4-port Mezzanine Adapter (e4145616e4140528).
- `nment_close` for the 4-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4143009).
- `nment_close` for the 2-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4140909).
- `nment_close` for the 2-Port Gigabit Ethernet PCI-Express Combo Adapter (e4143a161410a003).
- `nment_close` for the 2-Port Integrated Gigabit Ethernet PCI-Express Adapter (e4143a161410ed03).
- **scent_close** for the 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01).
- **phxent_close** for the 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020).
- `vent_close` for the 10 Gigabit Ethernet-SR PCI-X Adapter Device Driver (1410ba02) and the 10 Gigabit Ethernet-LR PCI_X Adapter Device Driver (1410bb02).

### *Data Transmission*

The output entry point transmits data using the specified network device.

The data to be transmitted is passed into the device driver by way of **mbuf** structures. The first **mbuf** structure in the chain must be of **M_PKTHDR** format. Multiple **mbuf** structures can be used to hold the frame. Link the **mbuf** structures using the **m_next** field of the **mbuf** structure.

Multiple packet transmits are supported with the mbufs being chained using the **m_nextpkt** field of the **mbuf** structure. The **m_pkthdr.len** field must be set to the total length of the packet. The device driver does *not* support mbufs from user memory (which have the **M_EXT** flag set).

On successful transmit requests, the device driver is responsible for freeing all the mbufs associated with the transmit request. If the device driver returns an error, the caller is responsible for the mbufs. If any of the chained packets can be transmitted, the transmit is considered successful and the device driver is responsible for all of the mbufs in the chain.

If the destination address in the packet is a broadcast address the **M_BCAST** flag in the **m_flags** field should be set prior to entering this routine. A broadcast address is defined as 0xFFFF FFFF FFFF. If the destination address in the packet is a multicast address the **M_MCAST** flag in the **m_flags** field should be set prior to entering this routine. A multicast address is defined as a non-individual address other than a broadcast address. The device driver keeps statistics based upon the **M_BCAST** and **M_MCAST** flags.

For packets that are shorter than the Ethernet minimum MTU size (60 bytes), the device driver pads them by adjusting the transmit length to the adapter so they can be transmitted as valid Ethernet packets.

Users are not notified by the device driver about the status of the transmission. Various statistics about data transmission are kept by the driver in the **ndd** structure. These statistics are part of the data returned by the **NDD_GET_STATS** control operation.

The output entry points are as follows:

- `bent_output` for the Gigabit Ethernet-SX Adapter Device Driver (e414a816).
- `elxent_output` for the 2-port 10GbE SR PCIe2 Adapter (a21910071410d003).
- `elxent_output` for the PCIe2 2-port 10GbE SFP+Copper Adapter (a21910071410d103).
- `elxent_output` for the 10GbE 4-port Mezzanine PCIe Adapter (a2191007df1033e7).
- `elxent_output` for the Int Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203).
- `elxent_output` for the Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203).
- **gxent_output** for the Gigabit Ethernet-SX PCI Adapter Device Driver (14100401)
- **goent_output** for the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802), the 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902), the 2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802), the 2-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver

(14108902), the 4-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14101103), and 4-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14106803), the 2-Port Gigabit Ethernet-SX PCI-Express Adapter Device Driver (14103f03), and the 2-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14104003).

- `hea_output` for the Host Ethernet Adapter Device Driver.
- **kent_output** for the PCI Ethernet Device Driver (22100020)
- `kngent_output` for the 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter Device Driver (1410eb02) and the 10 Gigabit Ethernet-LR PCI_X 2.0 DDR Adapter Device Driver (1410ec02).
- `ment_output` for the Gigabit Ethernet-SX Adapter Device Driver (14101403) and the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106703).
- `msnent_output` for the 10 Gb Ethernet PCI Express Dual Port Adapter (7710008077108001).
- `msnent_output` for the 10 Gb Ethernet-SR PCI Express Dual Port Adapter (771000801410b003).
- `musent_output` for the 4-Port Gigabit Ethernet PCI-Express Adapter (e414571614102004).
- `musent_output` for the 1GbE 4-port Mezzanine Adapter (e4145616e4140518) and 1GbE 4-port Mezzanine Adapter (e4145616e4140528).
- `nment_output` for the 4-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4143009).
- `nment_output` for the 2-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4140909).
- `nment_output` for the 2-Port Gigabit Ethernet PCI-Express Combo Adapter (e4143a161410a003).
- `nment_output` for the 2-Port Integrated Gigabit Ethernet PCI-Express Adapter (e4143a161410ed03).
- **phxent_output** for the 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020).
- **scent_output** for the 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01).
- `vent_output` for the 10 Gigabit Ethernet-SR PCI-X Adapter Device Driver (1410ba02) and the 10 Gigabit Ethernet-LR PCI_X Adapter Device Driver (1410bb02).

### *Data Reception*

When the Ethernet device drivers receive a valid packet from the network device, the device drivers call the **nd_receive** function that is specified in the **ndd_t** structure of the network device. The **nd_receive** function is part of a CDLI network demultiplexer. The packet is passed to the **nd_receive** function in the form of a mbuf.

The Ethernet device drivers can pass multiple packets to the **nd_receive** function by chaining the packets together using the **m_nextpkt** field of the **mbuf** structure. The **m_pkthdr.len** field must be set to the total length of the packet. If the source address in the packet is a broadcast address the **M_BCAST** flag in the **m_flags** field should be set. If the source address in the packet is a multicast address the **M_MCAST** flag in the **m_flags** field should be set.

When the device driver initially configures the device to discard all invalid frames. A frame is considered to be invalid for the following reasons:

- The packet is too short.
- The packet is too long.
- The packet contains a CRC error.
- The packet contains an alignment error.

If the asynchronous status for receiving invalid frames has been issued to the device driver, the device driver configures the device to receive bad packets as well as good packets. Whenever a bad packet is received by the driver, an asynchronous status block **NDD_BAD_PKTS** is created and delivered to the appropriate user. The user must copy the contents of the mbuf to another memory area. The user must not modify the contents of the mbuf or free the mbuf. The device driver has the responsibility of releasing the mbuf upon returning from **nd_status**.

Various statistics about data reception on the device are kept by the driver in the **ndd** structure. These statistics are part of the data returned by the **NDD_GET_STATS** and **NDD_GET_ALL_STATS** control operations.

There is no specified entry point for this function. The device informs the device driver of a received packet using an interrupt. Upon determining that the interrupt was the result of a packet reception, the device driver's interrupt handler invoke the **rx_handler** completion routine to perform the tasks mentioned above.

## Asynchronous Status

When a status event occurs on the device, the Ethernet device drivers build the appropriate status block and call the **nd_status** function that is specified in the **ndd_t** structure of the network device. The **nd_status** function is part of a CDLI network demuxer.

The following status blocks are defined for the Ethernet device drivers.

**Note:** The following device drivers support the Device Connected status block:

- Gigabit Ethernet-SX PCI Adapter Device Driver (14100401)
- Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802)
- 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902)
- 2-Port Gigabit Ethernet-SX PCI-X Adapter (14108802)
- 2-Port 10/100/1000 Base-TX PCI-X Adapter (14108902)
- 4-Port 10/100/1000 Base-TX PCI-X Adapter (14101103)
- 2-Port Gigabit Ethernet-SX PCI-Express Adapter Device Driver (14103f03)
- 2-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14104003)
- 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter (1410eb02)
- 10 Gigabit Ethernet-LR PCI-X 2.0 DDR Adapter (1410ec02)
- 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)

The PCI Ethernet Device Driver (22100020) supports the Bad Packets status block.

**Bad Packets**
When the a bad packet has been received by a device driver (and a user has requested bad packets), the following status block is returned by the device driver.

**code**
Set to `NDD_BAD_PKTS`.

**option[0]**
Specifies the error status of the packet. These error numbers are defined in **<sys/cdli_entuser.h>**.

**option[1]**
Pointer to the mbuf containing the bad packet.

**option[]**
The remainder of the status block can be used to return additional status information by the device driver.

**Note:** The user does *not* own the mbuf containing the bad packet. The user must copy the mbuf (and the status block information if necessary). The device driver frees the mbuf upon return from the **nd_status** function.

**Device Connected**
When the device is successfully connected to the network the following status block is returned by the device driver.

**code**
Set to `NDD_CONNECTED`.

**option[]**
The option fields are not used.

## Device Control Operations

The **ndd_ctl** entry point is used to provide device control functions.

### *NDD_GET_STATS Device Control Operation*

The **NDD_GET_STATS** command returns statistics concerning the network device. General statistics are maintained by the device driver in the **ndd_genstats** field in the **ndd_t** structure.

The **ndd_specstats** field in the **ndd_t** structure is a pointer to media-specific and device-specific statistics maintained by the device driver. Both sets of statistics are directly readable at any time by those users of the device that can access them. This command provides a way for any of the users of the device to access the general and media-specific statistics.

The *arg* and *length* parameters specify the address and length in bytes of the area where the statistics are to be written. The length specified *must* be the exact length of the general and media-specific statistics.

**Note:** The **ndd_speclen** field in the **ndd_t** structure plus the length of the **ndd_genstats_t** structure is the required length. The device-specific statistics might change with each new release of the operating system, but the general and media-specific statistics are not expected to change.

The user should pass in the **ent_ndd_stats_t** structure as defined in **sys/cdli_entuser.h**. The driver fails a call with a buffer smaller than the structure.

The statistics that are returned contain statistics obtained from the device. If the device is inoperable, the statistics that are returned do not contain the current device statistics. The copy of the **ndd_flags** field can be checked to determine the state of the device.

### *NDD_MIB_QUERY Device Control Operation*

The **NDD_MIB_QUERY** operation is used to determine which device-specific MIBs are supported on the network device. The *arg* and *length* parameters specify the address and length in bytes of a device-specific **MIB** structure. The device driver fills every member of that structure with a flag indicating the level of support for that member. The individual **MIB** variables that are not supported on the network device are set to **MIB_NOT_SUPPORTED**. The individual **MIB** variables that can only be read on the network device are set to **MIB_READ_ONLY**. The individual **MIB** variables that can be read and set on the network device are set to **MIB_READ_WRITE**. The individual **MIB** variables that can only be set (not read) on the network device are set to **MIB_WRITE_ONLY**. These flags are defined in the **/usr/include/sys/ndd.h** file.

The *arg* parameter specifies the address of the **Ethernet_all_mib** structure. This structure is defined in the **/usr/include/sys/Ethernet_mibs.h** file.

### *NDD_MIB_GET Device Control Operation*

The **NDD_MIB_GET** operation is used to get all MIBs on the specified network device. The *arg* and *length* parameters specify the address and length in bytes of the device specific MIB structure. The device driver sets any unsupported variables to zero (nulls for strings).

If the device supports the RFC 1229 receive address object, the corresponding variable is set to the number of receive addresses currently active.

The *arg* parameter specifies the address of the **Ethernet_all_mib** structure. This structure is defined in the **/usr/include/sys/Ethernet_mibs.h** file.

### *NDD_ENABLE_ADDRESS Device Control Operation*

The **NDD_ENABLE_ADDRESS** command enables the receipt of packets with an alternate (for example, multicast) address. The *arg* and *length* parameters specify the address and length in bytes of the alternate address to be enabled. The **NDD_ALTADDRS** flag in the **ndd_flags** field is set.

The device driver verifies that if the address is a valid multicast address. If the address is not a valid multicast address, the operation fails with an **EINVAL** error. If the address is valid, the driver adds it to its multicast table and enable the multicast filter on the adapter. The driver keeps a reference count for each individual address. Whenever a duplicate address is registered, the driver simply increments the

reference count of that address in its multicast table, no update of the adapter's filter is needed. There is a hardware limitation on the number of multicast addresses in the filter.

### *NDD_DISABLE_ADDRESS Device Control Operation*

The **NDD_DISABLE_ADDRESS** command disables the receiving packets with a specified alternate (for example, multicast) address. The *arg* and *length* parameters specify the address and length in bytes of the alternate address to be disabled. The **NDD_ALTADDRS** flag in the **ndd_flags** field is reset if this is the last alternate address.

The device driver verifies that if the address is a valid multicast address. If the address is not a valid multicast address, the operation fails with an **EINVAL** error. The device driver makes sure that the multicast address can be found in its multicast table. Whenever a match is found, the driver decrements the reference count of that individual address in its multicast table. If the reference count becomes 0, the driver deletes the address from the table and update the multicast filter on the adapter.

### *NDD_MIB_ADDR Device Control Operation*

The **NDD_MIB_ADDR** operation is used to get all the addresses for which the specified device accepts packets or frames.

The *arg* parameter specifies the address of the **ndd_mib_addr_t** structure. The length parameter specifies the length of the structure with the appropriate number of **ndd_mib_addr_t.mib_addr** elements. This structure is defined in the **/usr/include/sys/ndd.h** file. If the length is less than the length of the **ndd_mib_addr_t** structure, the device driver returns **EINVAL**. If the structure is not large enough to hold all the addresses, the addresses that fit are still placed in the structure. The **ndd_mib_addr_t.count** field is set to the number of addresses returned and `E2BIG` is returned.

One of the following address types is returned:

- Device physical address (or alternate address specified by user)
- Broadcast addresses
- Multicast addresses

### *NDD_CLEAR_STATS Device Control Operation*

The counters kept by the device are zeroed.

### *NDD_GET_ALL_STATS Device Control Operation*

The **NDD_GET_ALL_STATS** operation is used to gather all the statistics for the specified device.

The **arg** parameter specifies the address of the statistics structure for the particular device type. The following structures are available:

- The **kent_all_stats_t** structure is available for the PCI Ethernet Adapter Device Driver (22100020), and is defined in the **cdli_entuser.h** include file.
- The **phxent_all_stats_t** structure is available for the 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020), and is defined in the device-specific **cdli_entuser.phxent.h** include file.
- The **scent_all_stats_t** structure is available for the 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01), and is defined in the device-specific **cdli_entuser.scent.h** include file.
- The **gxent_all_stats_t** structure is available for the Gigabit Ethernet-SX PCI Adapter Device Driver (14100401), and is defined in the device-specific **cdli_entuser.gxent.h** include file.
- The **goent_all_stats_t** structure is available for the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802), the 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902), the 2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802), the 2-Port 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14108902), the 4-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14101103), and the 4-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14106803), the 2-Port Gigabit Ethernet-SX PCI-Express Adapter Device Driver (14103f03), and the 2-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14104003), and is defined in the device-specific **cdli_entuser.goent.h** include file.

- The `vent_all_stats_t` structure is available for the 10 Gigabit Ethernet-SR PCI-X Adapter Device Driver (1410ba02) and the 10 Gigabit Ethernet-LR PCI_X Adapter Device Driver (1410bb02), and is defined in the device-specific `cdli_entuser.vent.h` include file.
- The `bent_all_stats_t` structure is available for the Gigabit Ethernet-SX Adapter Device Driver (e414a816), and is defined in the device-specific `cdli_entuser.bent.h` include file.
- The `ment_all_stats_t` structure is available for the Gigabit Ethernet-SX Adapter Device Driver (14101403) and the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106703), and is defined in the device-specific `cdli_entuser.ment.h` include file.
- The `kngent_all_stats_t` structure is available for the 10 Gigabit Ethernet-SR Adapter Device Driver (1410eb02) and the 10 Gigabit Ethernet-LR PCI-X 2.0 DDR Adapter (1410ec02), and is defined in the device-specific `cdli_entuser.kngent.h` include file.
- The `hea_all_stats_t` structure is available for the Host Ethernet Adapter Device Driver, and is defined in the `cdli_entuser.hea.h` include file.

The statistics that are returned contain statistics obtained from the device. If the device is inoperable, the statistics that are returned do not contain the current device statistics. The copy of the **ndd_flags** field can be checked to determine the state of the device.

### *NDD_ENABLE_MULTICAST Device Control Operation*

The **NDD_ENABLE_MULTICAST** command enables the receipt of packets with any multicast (or group) address.

The *arg* and *length* parameters are not used. The **NDD_MULTICAST** flag in the **ndd_flags** field is set.

### *NDD_DISABLE_MULTICAST Device Control Operation*

The **NDD_DISABLE_MULTICAST** command disables the receipt of *all* packets with multicast addresses and only receives those packets whose multicast addresses were specified using the **NDD_ENABLE_ADDRESS** command.

The *arg* and *length* parameters are not used. The **NDD_MULTICAST** flag in the **ndd_flags** field is reset only after the reference count for multicast addresses has reached zero.

### *NDD_PROMISCUOUS_ON Device Control Operation*

The **NDD_PROMISCUOUS_ON** command turns on promiscuous mode. The *arg* and *length* parameters are not used.

When the device driver is running in promiscuous mode, all network traffic is passed to the network demultiplexer. When the Ethernet device driver receives a valid packet from the network device, the Ethernet device driver calls the **nd_receive** function that is specified in the **ndd_t** structure of the network device. The **NDD_PROMISC** flag in the **ndd_flags** field is set. Promiscuous mode is considered to be valid packets only. See the **NDD_ADD_STATUS** command for information about how to request support for bad packets.

The device driver maintains a reference count on this operation. The device driver increments the reference count for each operation. When this reference count is equal to one, the device driver issues commands to enable the promiscuous mode. If the reference count is greater than one, the device driver does not issue any commands to enable the promiscuous mode.

### *NDD_PROMISCUOUS_OFF Device Control Operation*

The **NDD_PROMISCUOUS_OFF** command terminates promiscuous mode.

The *arg* and *length* parameters are not used. The **NDD_PROMISC** flag in the **ndd_flags** field is reset.

The device driver maintains a reference count on this operation. The device driver decrements the reference count for each operation. When the reference count is not equal to zero, the device driver does not issue commands to disable the promiscuous mode. After the reference count for this operation is equal to zero, the device driver issues commands to disable the promiscuous mode.

### *NDD_DISABLE_ADAPTER Device Control Operation*

The **NDD_DISABLE_ADAPTER** operation is used by Etherchannel to disable the adapter so that it cannot transmit or receive data.

**Note:** This device control operation is not supported on the PCI Ethernet Adapter Device Driver (22100020).

During this operation the **NDD_RUNNING** and **NDD_LIMBO** flags are cleared and the adapter is reset. The *arg* and *len* parameters are not used.

### *NDD_ENABLE_ADAPTER Device Control Operation*

The **NDD_ENABLE_ADAPTER** operation is used by Etherchannel to return the adapter to a running state so it can transmit and receive data. During this operation the adapter is started and the **NDD_RUNNING** flag is set.

**Note:** This device control operation is not supported on the PCI Ethernet Adapter Device Driver (22100020).

The *arg* and *len* parameters are not used.

### *NDD_SET_LINK_STATUS Device Control Operation*

The **NDD_SET_LINK_STATUS** operation is used by Etherchannel to pass the driver a function pointer and argument that will subsequently be called by the driver whenever the link status changes.

**Note:** This device control operation is not supported on the PCI Ethernet Adapter Device Driver (22100020).

The *arg* parameter contains a pointer to a **ndd_sls_t** structure, and the *len* parameter contains the length of the **ndd_sls_t** structure.

### *NDD_SET_MAC_ADDR Device Control Operation*

The **NDD_SET_NAC_ADDR** operation is used by Etherchannel to set the adapters MAC address at runtime.

**Note:** This device control operation is not supported on the PCI Ethernet Adapter Device Driver (22100020).

The MAC address set by this ioctl is valid until another **NDD_SET_MAC_ADDR** call is made with a new address or when the adapter is closed. If the adapter is closed, the previously configured MAC address. The MAC address configured with the ioctl supersedes any alternate address that might have been configured.

The *arg* argument is `char [6]`, representing the MAC address that is configured on the adapter. The *len* argument is 6.

## Trace

For LAN device drivers, trace points enable error monitoring as well as tracking packets as they move through the driver.

The drivers issue trace points for some or all of the following conditions:

- Beginning and ending of main functions in the main path
- Error conditions
- Beginning and ending of each function that is tracking buffers outside of the main path
- Debugging purposes (These trace points are only enabled when the driver is compiled with **-DDEBUG** turned on, and therefore the driver can contain as many of these trace points as necessary.)

The existing Ethernet device drivers each have either three or four trace points. The Trace Hook IDs the PCI Ethernet Adapter Device Driver (22100020) is defined in the **sys/cdli_entuser.h** file. Other drivers have defined local **cdli_entuser.*driver*.h** files with the Trace Hook definitions. For more information, see “Debug and Performance Tracing” on page 388.

Following is a list of trace hooks (and location of definition file) for the existing Ethernet device drivers.

### *PCI Ethernet Adapter Device Driver (22100020)*

The definition file and trace hook ID of PCI Ethernet Adapter Device Driver (22100020) are as follows:

Definition file: **cdli_entuser.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | -2A4 |
| Receive | -2A5 |
| Other | -2A6 |

### *10/100 Mbps Ethernet PCI Adapter Device Driver (23100020)*

Following are the definition files and the trace hook IDs for 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020) device driver.

Definition file: **cdli_entuser.phxent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | -2E6 |
| Receive | -2E7 |
| Other | -2E8 |

### *10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01)*

Following are the definition files and the trace hook IDs for 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) device driver.

Definition file: **cdli_entuser.scent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | -470 |
| Receive | -471 |
| Other | -472 |

### *Gigabit Ethernet-SX PCI Adapter Device Driver (14100401)*

The definition file and trace hook IDs of Gigabit Ethernet-SX PCI adapter device driver (14100401) are as follows:

Definition file: **cdli_entuser.gxent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | -2EA |
| Receive | -2EB |
| Other | -2EC |

### *Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802), 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902), 2-Port Gigabit Ethernet-SX PCI-X*

### *Adapter (14108802), 2-Port 10/100/1000 Base-TX PCI-X Adapter (14108902), 4-Port 10/100/1000 Base-TX PCI-X Adapter (14101103), 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803), 2-Port Gigabit Ethernet-SX PCI-Express Adapter (14103f03), and 2-Port 10/100/1000 Base-TX PCI-Express Adapter (14104003)*

The definition file and trace hook IDs are as follows:

Definition file: **cdli_entuser.goent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | -473 |
| Receive | -474 |
| Other | -475 |

The device driver also has the following trace points to support the **netpmon** program:

| Item | Description |
|---|---|
| **WQUE** | An output packet has been queued for transmission. |
| **WEND** | The output of a packet is complete. |
| **RDAT** | An input packet has been received by the device driver. |
| **RNOT** | An input packet has been given to the demuxer. |
| **REND** | The demultiplexer has returned. |

### *10 Gigabit Ethernet-SR PCI-X Adapter (1410ba02) and 10 Gigabit Ethernet-LR PCI_X Adapter (1410bb02)*

The definition file and trace hook IDs of 10 Gigabit Ethernet-SR PCI-X adapter (1410ba02) and 10 Gigabit Ethernet-LR PCI_X adapter (1410bb02) are as follows:

Definition file: **cdli_entuser.vent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | -598 |
| Receive | -599 |
| Other | -59A |

The device driver also has the following trace points to support the **netpmon** program:

| Item | Description |
|---|---|
| **WQUE** | An output packet has been queued for transmission. |
| **WEND** | The output of a packet is complete. |
| **RDAT** | An input packet has been received by the device driver. |
| **RNOT** | An input packet has been given to the demuxer. |
| **REND** | The demultiplexer has returned. |

### *Gigabit Ethernet-SX Adapter Device Driver (e414a816)*

The definition file and trace hook IDs of Gigabit Ethernet-SX adapter device driver (e414a816) are as follows:

Definition file: **cdli_entuser.bent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | -5B2 |
| Receive | -5B3 |
| Other | -5B4 |

Definition file: **cdli_entuser.kngent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | -4a1 |
| Receive | -4a2 |
| Other | -4a3 |

The device driver also has the following trace points to support the `netpmon` program:

**WQUE**
An output packet has been queued for transmission.

**WEND**
The output of a packet is complete.

**RDAT**
An input packet has been received by the device driver.

**RNOT**
An input packet has been given to the demuxer.

**REND**
The demultiplexer has returned.

### *Gigabit Ethernet-SX Adapter Device Driver (14101403) and Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106703)*

The definition file and trace hook IDs of Gigabit Ethernet-SX adapter device driver (14101403) and Gigabit Ethernet-SX PCI-X adapter device driver (14106703) are as follows:

Definition file: **cdli_entuser.ment.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | -4C5 |
| Receive | -4C6 |
| Other | -4C7 |

### *Host Ethernet Adapter Device Driver*

The definition file and trace hook IDs of Host Ethernet adapter device driver are as follows:

Definition file: **cdli_entuser.hea.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x5df |
| Receive | 0x5e0 |
| Other | 0x5e1 |

### *4-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e414571614102004)*

The definition file and trace hook IDs of 4-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e414571614102004) are as follows:

Definition file: **cdli_entuser.musent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x664 |
| Receive | 0x665 |
| Other | 0x666 |

### *1GbE 4-port Mezzanine Adapter (e4145616e4140518) and 1GbE 4-port Mezzanine Adapter (e4145616e4140528) Device Driver*

The definition file and trace hook IDs of 1GbE 4-port Mezzanine adapter (e4145616e4140518) and 1GbE 4-port Mezzanine adapter (e4145616e4140528) device driver are as follows:

Definition file: **cdli_entuser.musent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x664 |
| Receive | 0x665 |
| Other | 0x666 |

### *4-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a16e4143009)*

The definition file and trace hook IDs of 4-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a16e4143009) are as follows:

Definition file: **cdli_entuser.nment.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x660 |
| Receive | 0x661 |
| Other | 0x662 |

### *2-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a16e4140909)*

The definition file and trace hook IDs of 2-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a16e4140909) are as follows:

Definition file: **cdli_entuser.nment.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x660 |
| Receive | 0x661 |
| Other | 0x662 |

### *2-Port Gigabit Ethernet PCI-Express Combo Adapter Device Driver (e4143a161410a003)*

The definition file and trace hook IDs of 2-Port Gigabit Ethernet PCI-Express combo adapter sevice driver (e4143a161410a003) are as follows:

Definition file: **cdli_entuser.nment.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x660 |
| Receive | 0x661 |
| Other | 0x662 |

### *2-Port Integrated Gigabit Ethernet PCI-Express adapter device driver (e4143a161410ed03)*

The definition file and trace hook IDs of 2-Port Integrated Gigabit Ethernet PCI-Express adapter device driver (e4143a161410ed03) are as follows:

Definition file: **cdli_entuser.nment.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x660 |
| Receive | 0x661 |
| Other | 0x662 |

### *10 Gb Ethernet PCI Express Dual Port Adapter Device Driver (7710008077108001)*

The definition file and trace hook IDs of 10 Gb Ethernet PCI Express Dual Port adapter device driver (7710008077108001) are as follows:

Definition file: **cdli_entuser.msnent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x63D |
| Receive | 0x63E |
| Other | 0x63F |

### *10 Gb Ethernet-SR PCI Express Dual Port Adapter Device Driver (771000801410b003)*

The definition file and trace hook IDs of 10 Gb Ethernet-SR PCI Express Dual Port adapter device driver (771000801410b003) are as follows:

Definition file: **cdli_entuser.msnent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x63D |
| Receive | 0x63E |
| Other | 0x63F |

### *2-port 10GbE SR PCIe2 Adapter Device Driver (a21910071410d003)*

The definition file and trace hook IDs of 2-port 10 GbE SR PCIe2 Adapter device driver (a21910071410d003) are as follows:

Definition file: **cdli_entuser.elxent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x65D |
| Receive | 0x65E |
| Other | 0x65F |

### *PCIe2 2-port 10GbE SFP+Copper Adapter Device Driver (a21910071410d103)*

The definition file and trace hook IDs of PCIe2 2-port 10GbE SFP+Copper adapter device driver (a21910071410d103) are as follows:

Definition file: **cdli_entuser.elxent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x65D |
| Receive | 0x65E |
| Other | 0x65F |

### *10GbE 4-port Mezzanine Adapter Device Driver (a2191007df1033e7)*

The definition file and trace hook IDs of 10GbE 4-port Mezzanine adapter device driver (a2191007df1033e7) are as follows:

Definition file: **cdli_entuser.elxent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x65D |
| Receive | 0x65E |
| Other | 0x65F |

### *Int Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203) Device Driver*

The definition file and trace hook IDs of Int Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203) device driver are as follows:

Definition file: **cdli_entuser.elxent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x65D |
| Receive | 0x65E |
| Other | 0x65F |

### *Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203) Device Driver*

The definition file and trace hook IDs of Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203) device driver are as follows:

Definition file: **cdli_entuser.elxent.h**

Trace Hook IDs:

| Item | Description |
|---|---|
| Transmit | 0x65D |
| Receive | 0x65E |
| Other | 0x65F |

## Error Logging

You are responsible for implementing the proper level of thresholding in the device driver code. The default size of the error log is 1 MB.

For error logging information, see "Error Logging" on page 385.

### *PCI Ethernet Adapter Device Driver (22100020)*

The Error IDs for the PCI Ethernet Adapter Device Driver (22100020) are as follows:

**ERRID_KENT_ADAP_ERR**
Indicates that the adapter is not responding to initialization commands. User intervention is necessary to fix the problem.

**ERRID_KENT_RCVRY**
Indicates that the device driver detected a temporary adapter error requiring that it enter network recovery mode. It has reset the adapter in an attempt to fix the problem.

**ERRID_KENT_TX_ERR**
Indicates that the device driver has detected a transmission error. User intervention is not required unless the problem persists.

**ERRID_KENT_PIO**
Indicates that the device driver has detected a program IO error. The device driver was unable to fix the problem. User intervention is necessary to fix the problem.

**ERRID_KENT_DOWN**
Indicates that the device driver has shut down the adapter due to an unrecoverable error. The adapter is no longer functional due to the error. The error that caused the device to shut down is error logged immediately before this error log entry. User intervention is necessary to fix the problem.

### *10/100 Mbps Ethernet PCI Adapter Device Driver (23100020)*

The Error IDs for the 10/100 Mbps Ethernet PCI Adapter Device Driver (23100020) are as follows:

**ERRID_PHXENT_ADAP_ERR**
Indicates that the adapter is not responding to initialization commands. User-intervention is necessary to fix the problem.

**ERRID_PHXENT_ERR_RCVRY**
Indicates that the device driver detected a temporary adapter error requiring that it enter network recovery mode. It has reset the adapter in an attempt to fix the problem.

**ERRID_PHXENT_TX_ERR**
Indicates that the device driver has detected a transmission error. User-intervention is not required unless the problem persists.

**ERRID_PHXENT_PIO**
Indicates that the device driver has detected a program IO error. The device driver was unable to fix the problem. User intervention is necessary to fix the problem.

**ERRID_PHXENT_DOWN**
Indicates that the device driver has shutdown the adapter due to an unrecoverable error. The adapter is no longer functional due to the error. The error that caused the device shutdown is error logged immediately before this error log entry. User intervention is necessary to fix the problem.

**ERRID_PHXENT_EEPROM_ERR**
Indicates that the device driver is in a defined state due to an invalid or bad EEPROM. The device driver does not become available. Hardware support should be contacted.

**ERRID_PHXENT_EEPROM2_ERR**
Indicates that the device driver is in a defined state due to an invalid or bad EEPROM. The device driver does not become available. Hardware support should be contacted.

**ERRID_PHXENT_CLOSE_ERR**
Indicates that an application is holding a private receive mbuf owned by the device driver during a close operation. User intervention is not required.

**ERRID_PHXENT_LINK_ERR**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is again established, the device driver logs **ERRID_PHXENT_ERR_RCVRY**. User intervention is necessary to fix the problem.

### *Gigabit Ethernet-SX PCI Adapter Device Driver (14100401)*
The Error IDs for the Gigabit Ethernet-SX PCI Adapter Device Driver (14100401) are as follows:

**ERRID_GXENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to fix the problem.

**ERRID_GXENT_CMD_ERR**
Indicates that the device driver has detected an error while issuing commands to the adapter. The device driver enters an adapter recovery mode where it attempts to recover from the error. If the device driver is successful, it logs **ERRID_GXENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_GXENT_DOWNLOAD_ERR**
Indicates that an error occurred while downloading firmware to the adapter. User intervention is necessary to fix the problem.

**ERRID_GXENT_EEPROM_ERR**
Indicates that an error occurred while reading the adapter EEPROM. User intervention is necessary to fix the problem.

**ERRID_GXENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is again established, the device driver logs **ERRID_GXENT_RCVRY_EXIT**. User intervention is necessary to fix the problem.

**ERRID_GXENT_RCVRY_EXIT**
Indicates that a temporary error (link down, command error, or transmission error) has been corrected.

**ERRID_GXENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_GXENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_GXENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to fix the problem.

### *10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01)*

The Error IDs for the 10/100 Mbps Ethernet PCI Adapter II Device Driver (1410ff01) are as follows:

**ERRID_SCENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to fix the problem.

**ERRID_SCENT_PIO_ERR**
Indicates that the device driver has detected a program IO error. The device driver was unable to fix the problem. User intervention is necessary to fix the problem.

**ERRID_SCENT_EEPROM_ERR**
Indicates that an error occurred while reading the adapter EEPROM. User intervention is necessary to fix the problem.

**ERRID_SCENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is again established, the device driver logs **ERRID_SCENT_RCVRY_EXIT**. User intervention is necessary to fix the problem.

**ERRID_SCENT_RCVRY_EXIT**
Indicates that a temporary error (link down, command error, or transmission error) has been corrected.

**ERRID_SCENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_SCENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_SCENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to fix the problem.

### *Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802) and Other Device Drivers*

This section lists the error IDs of the following device drivers:

- Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106802)
- 10/100/1000 Base-T Ethernet PCI-X Adapter Device Driver (14106902)
- 2-Port Gigabit Ethernet-SX PCI-X Adapter Device Driver (14108802)
- 2-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14108902)
- 4-Port 10/100/1000 Base-TX PCI-X Adapter Device Driver (14101103)
- 4-Port 10/100/1000 Base-TX PCI-Express Adapter Device Driver (14106803)
- 2-Port Gigabit Ethernet-SX PCI-Express Adapter (14103f03)
- 2-Port 10/100/1000 Base-TX PCI-Express Adapter (14104003)

**ERRID_GOENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to fix the problem.

**ERRID_GOENT_PIO_ERR**
Indicates that the device driver has detected a program I/O error. The device driver was unable to fix the problem. User intervention is necessary to fix the problem.

**ERRID_GOENT_EEPROM_ERR**
Indicates that an error occurred while reading the adapter EEPROM. User intervention is necessary to fix the problem.

**ERRID_GOENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is again

established, the device driver logs **ERRID_GOENT_RCVRY_EXIT**. User intervention is necessary to fix the problem.

**ERRID_GOENT_RCVRY_EXIT**
Indicates that a temporary error (link down, command error, or transmission error) has been corrected.

**ERRID_GOENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_GOENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_GOENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to fix the problem.

### *10 Gigabit Ethernet-SR PCI-X Adapter (1410ba02) and 10 Gigabit Ethernet-LR PCI_X Adapter (1410bb02)*

The error IDs for the 10 Gigabit Ethernet-SR PCI-X Adapter (1410ba02) and 10 Gigabit Ethernet-LR PCI_X Adapter (1410bb02) are as follows:

**ERRID_VENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to fix the problem.

**ERRID_VENT_PIO_ERR**
Indicates that the device driver has detected a program I/O error. The device driver was unable to fix the problem. User intervention is necessary to fix the problem.

**ERRID_VENT_EEPROM_ERR**
Indicates that an error occurred while reading the adapter EEPROM. User intervention is necessary to fix the problem.

**ERRID_VENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is again established, the device driver logs **ERRID_VENT_RCVRY_EXIT**. User intervention is necessary to fix the problem.

**ERRID_VENT_RCVRY_EXIT**
Indicates that a temporary error (link down, command error, or transmission error) has been corrected.

**ERRID_VENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_VENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_VNT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to fix the problem.

### *Gigabit Ethernet-SX Adapter Device Driver (e414a816)*

The Error IDs for the Gigabit Ethernet-SX Adapter Device Driver (e414a816) are as follows:

**ERRID_BENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to fix the problem.

**ERRID_BENT_DOWNLOAD_ERR**
Indicates that an error occurred while downloading firmware to the adapter. User intervention is necessary to fix the problem.

**ERRID_BENT_EEPROM_ERR**
Indicates that an error occurred while reading the adapter EEPROM. User intervention is necessary to fix the problem.

**ERRID_BENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is again established, the device driver logs **ERRID_BENT_RCVRY_EXIT**. User intervention is necessary to fix the problem.

**ERRID_BENT_RCVRY_EXIT**
Indicates that a temporary error (link down, or transmission error) was corrected.

**ERRID_BENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_BENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

### *Gigabit Ethernet-SX Adapter Device Driver (14101403) and Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106703)*

The Error IDs for the Gigabit Ethernet-SX Adapter Device Driver (14101403) and the Gigabit Ethernet-SX PCI-X Adapter Device Driver (14106703) are as follows:

**ERRID_MENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to fix the problem.

**ERRID_MENT_DOWNLOAD_ERR**
Indicates that an error occurred while downloading firmware to the adapter. User intervention is necessary to fix the problem.

**ERRID_MENT_EEPROM_ERR**
Indicates that an error occurred while reading the adapter EEPROM. User intervention is necessary to fix the problem.

**ERRID_MENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is again established, the device driver logs **ERRID_MENT_RCVRY_EXIT**. User intervention is necessary to fix the problem.

**ERRID_MENT_RCVRY_EXIT**
Indicates that a temporary error (link down, or transmission error) was corrected.

**ERRID_MENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_MENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

### *10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter (1410eb02) and 10 Gigabit Ethernet-LR PCI_X 2.0 DDR Adapter (1410ec02)*

The error IDs for the 10 Gigabit Ethernet-SR PCI-X 2.0 DDR Adapter (1410eb02) and 10 Gigabit Ethernet-LR PCI_X 2.0 DDR Adapter (1410ec02) are as follows:

**ERRID_KNGENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to fix the problem.

**ERRID_KNGENT_PIO_ERR**
Indicates that the device driver has detected a program I/O error. The device driver was unable to fix the problem. User intervention is necessary to fix the problem.

**ERRID_KNGENT_EEPROM_ERR**
Indicates that an error occurred while reading the adapter EEPROM. User intervention is necessary to fix the problem.

**ERRID_KNGENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is again established, the device driver logs ERRID_KNGENT_RCVRY_EXIT. User intervention is necessary to fix the problem.

**ERRID_KNGENT_RCVRY_EXIT**
Indicates that a temporary error (link down, command error, or transmission error) has been corrected.

**ERRID_KNGENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs ERRID_KNGENT_RCVRY_EXIT. User intervention is not necessary for this error unless the problem persists.

**ERRID_KNGENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected an error during an attempt to recover from a PCI bus error. User intervention is necessary to fix the problem.

### *Host Ethernet Adapter device driver*
The error IDs for the Host Ethernet adapter device driver are as follows:

**ERRID_KNGENT_EEH_SERVICE_ERR**
Indicates that the adapter had a permanent hardware failure. This error is irrecoverable. This error is logged when a notification-event-queue event, for example, ADAPTER_MALFUNCTION, occurred. User intervention is necessary to fix the problem.

**ERRID_HEA_ENS_PORT_ERR**
Indicates that the external network logical switch has a temporary port error or a link down. The adapter and other ports of the adapter, if any, remain available. This event is notified to the Host Ethernet Adapter device driver through a notification-event-queue event of the HEA_ENS_PORT_MALFUNCTION type. User intervention is necessary to fix the problem.

**ERRID_HEA_LINK_DOWN**
Indicates that the adapter has detected that the Ethernet link is down. This event is notified to the Host Ethernet Adapter device driver through a notification-event-queue event that is PORT_STATE_CHANGE. When the link is re-established, the device driver logs the ERRID_HEA_ERR_RCVRY_EXIT error. User intervention is necessary to fix the problem.

**ERRID_HEA_ERR_RCVRY_EXIT**
Indicates that the adapter encountered a temporary error (TX_ERR or LINK_DOWN) that halted network activity and the problem has been resolved.

**ERRID_HEA_TX_ERR**
Indicates that the device driver has detected a transmission error (transmit timeout). If the recovery is successful, the driver logs the ERRID_HEA_ERR_RCVRY_EXIT error; otherwise the ERRID_HEA_ADAP_ERR error is logged. User intervention is not required unless the ERRID_HEA_ADAP_ERR error is also logged.

### *4-Port Gigabit Ethernet PCI-Express Adapter (e414571614102004)*
The error IDs for the 4-Port Gigabit Ethernet PCI-Express adapter device driver (e414571614102004) are as follows:

**ERRID_MUSENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshoot the problem.

**ERRID_MUSENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link

is established again, the device driver logs **ERRID_MUSENT_RCVRY_EXIT**. User intervention is necessary to troubleshoot the problem.

**ERRID_MUSENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected.

**ERRID_MUSENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_MUSENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_MUSENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected an error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshoot the problem.

**ERRID_MUSENT_EEH_ENTRY**
Indicates that the device driver detected enhance error handling (EEH) freeze condition.

### *1GbE 4-port Mezzanine Adapter (e4145616e4140518) and 1GbE 4-port Mezzanine Adapter (e4145616e4140528)*

The error IDs for the 1GbE 4-port Mezzanine Adapter Device Driver (e4145616e4140518) and and 1GbE 4-port Mezzanine Adapter Device Driver (e4145616e4140528) are as follows:

**ERRID_MUSENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshoot the problem.

**ERRID_MUSENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_MUSENT_RCVRY_EXIT**. User intervention is necessary to troubleshoot the problem.

**ERRID_MUSENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected.

**ERRID_MUSENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_MUSENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_MUSENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshoot the problem.

**ERRID_MUSENT_EEH_ENTRY**
Indicates that the device driver found Enhance Error Handling (EEH) freeze condition.

### *4-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4143009)*

The error IDs for the 4-Port Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a16e4143009) are as follows:

**ERRID_NMENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshoot the problem.

**ERRID_NMENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_NMENT_RCVRY_EXIT**. User intervention is necessary to troubleshoot the problem.

**ERRID_NMENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected.

**ERRID_NMENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_NMENT_RCVRY_EXIT** . User intervention is not necessary for this error unless the problem persists.

**ERRID_NMENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshoot the problem.

**ERRID_NMENT_EEH_ENTRY**
Indicates that the device driver found enhance error handling (EEH) freeze condition.

**ERRID_NMENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

### *2-Port Gigabit Ethernet PCI-Express Adapter (e4143a16e4140909)*

The error IDs for the 2-Port Gigabit Ethernet PCI-Express adapter device driver (e4143a16e4140909) are as follows:

**ERRID_NMENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshooting the problem.

**ERRID_NMENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_NMENT_RCVRY_EXIT**. User intervention is necessary to troubleshooting the problem.

**ERRID_NMENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected.

**ERRID_NMENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_NMENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_NMENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshooting the problem.

**ERRID_NMENT_EEH_ENTRY**
Indicates that the device driver found enhanced error handling (EEH) freeze condition.

**ERRID_NMENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

### *2-Port Gigabit Ethernet PCI-Express Combo Adapter (e4143a161410a003)*

The error IDs for the 2-Port Gigabit Ethernet PCI-Express Combo Adapter Device Driver (e4143a161410a003) are as follows:

**ERRID_NMENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshooting the problem.

**ERRID_NMENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link

is established again , the device driver logs **ERRID_NMENT_RCVRY_EXIT**. User intervention is necessary to troubleshooting the problem.

**ERRID_NMENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected.

**ERRID_NMENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_NMENT_RCVRY_EXIT** . User intervention is not necessary for this error unless the problem persists.

**ERRID_NMENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshooting the problem.

**ERRID_NMENT_EEH_ENTRY**
Indicates that the device driver found enhanced error handling (EEH) freeze condition.

**ERRID_NMENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

### *2-Port Integrated Gigabit Ethernet PCI-Express Adapter (e4143a161410ed03)*

The error IDs for the 2-Port Integrated Gigabit Ethernet PCI-Express Adapter Device Driver (e4143a161410ed03) are as follows:

**ERRID_NMENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshoot the problem.

**ERRID_NMENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_NMENT_RCVRY_EXIT**. User intervention is necessary to troubleshoot the problem.

**ERRID_NMENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected.

**ERRID_NMENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_NMENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_NMENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshoot the problem.

**ERRID_NMENT_EEH_ENTRY**
Indicates that the device driver found enhanced error handling (EEH) freeze condition.

**ERRID_NMENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

### *10 Gb Ethernet PCI Express Dual Port Adapter (7710008077108001)*

The error IDs for the 10 Gb Ethernet PCI Express Dual Port adapter device driver (7710008077108001) are as follows:

**ERRID_MSNENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshooting the problem.

**ERRID_MSNENT_PIO_ERR**
Indicates that the device driver has detected a program IO error. The device driver was unable to troubleshooting the problem. User intervention is necessary to troubleshooting the problem.

**ERRID_MSNENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_MSNENT_RCVRY_EXIT**. User intervention is necessary to troubleshooting the problem.

**ERRID_MSNENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected.

**ERRID_MSNENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_MSNENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_MSNENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshooting the problem.

**ERRID_MSNENT_EEH_ENTRY**
Indicates that the device driver found enhanced error handling (EEH) freeze condition.

**ERRID_MSNENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

**ERRID_MSNENT_MPIFW_ERR**
Indicates a MPI firmware error.

**ERRID_MSNENT_FLASH_ERR**
Indicates a flash error.

### *10 Gb Ethernet-SR PCI Express Dual Port Adapter (771000801410b003)*

The error IDs for the 10 Gb Ethernet-SR PCI Express Dual Port adapter device driver (771000801410b003) are as follows:

**ERRID_MSNENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshooting the problem.

**ERRID_MSNENT_PIO_ERR**
Indicates that the device driver has detected a program IO error. The device driver was unable to troubleshooting the problem. User intervention is necessary to troubleshooting the problem.

**ERRID_MSNENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_MSNENT_RCVRY_EXIT**. User intervention is necessary to troubleshooting the problem.

**ERRID_MSNENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) has been corrected.

**ERRID_MSNENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_MSNENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_MSNENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshooting the problem.

**ERRID_MSNENT_EEH_ENTRY**
Indicates that the device driver found enhanced error handling (EEH) freeze condition.

**ERRID_MSNENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

**ERRID_MSNENT_MPIFW_ERR**
Indicates a MPI firmware error.

**ERRID_MSNENT_FLASH_ERR**
Indicates a flash error.

### *2-port 10GbE SR PCIe2 Adapter (a21910071410d003)*

The error IDs for the 2-port 10GbE SR PCIe2 Adapter Device Driver (a21910071410d003) are as follows:

**ERRID_ELXENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshoot the problem.

**ERRID_ELXENT_PIO_ERR**
Indicates that the device driver has detected a program IO error. The device driver was unable to troubleshoot the problem. User intervention is necessary to troubleshoot the problem.

**ERRID_ELXENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_ELXENT_RCVRY_EXIT**. User intervention is necessary to troubleshoot the problem

**ERRID_ELXENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected

**ERRID_ELXENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_ELXENT_RCVRY_EXIT** . User intervention is not necessary for this error unless the problem persists.

**ERRID_ELXENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshoot the problem.

**ERRID_ELXENT_VPD_ERR**
Indicates that the device driver could not able to get VPD (Vital Product Data) from the device, even though device firmware set VPD capabilities.

**ERRID_ELXENT_EEH_ENTRY**
Indicates that the device driver found enhanced error handling (EEH) freeze condition.

**ERRID_ELXENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

**ERRID_ELXENT_MBX_ERR**
Indicates that the device driver got an error when device could not able to complete the request asked by the driver.

**ERRID_ELXENT_EEPROM_ERR**
Indicates that the device driver is in a defined state due to an invalid or bad EEPROM. The device driver does not become available. Hardware support should be contacted.

### *PCIe2 2-port 10GbE SFP+Copper Adapter (a21910071410d103)*

The error IDs for the PCIe2 2-port 10GbE SFP+Copper Adapter Device Driver (a21910071410d103) are as follows:

**ERRID_ELXENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_PIO_ERR**
Indicates that the device driver has detected a program IO error. The device driver was unable to troubleshooting the problem. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_ELXENT_RCVRY_EXIT**. User intervention is necessary to troubleshooting the problem

**ERRID_ELXENT_RCVRY_EXIT**
Indicates that a temporary error (link down, command error, or transmission error) is corrected

**ERRID_ELXENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it log**sERRID_ELXENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_ELXENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_VPD_ERR**
Indicates that the device driver could not able to get VPD (Vital Product Data) from the device, even though device firmware set VPD capabilities.

**ERRID_ELXENT_EEH_ENTRY**
Indicates that the device driver found EEH freeze condition.

**ERRID_ELXENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from enhanced error handling (EEH) freeze condition.

**ERRID_ELXENT_MBX_ERR**
Indicates that the device driver got an error when device could not able to complete the request asked by the driver.

**ERRID_ELXENT_EEPROM_ERR**
Indicates that the device driver is in a defined state due to an invalid or bad electronically erasable programmable read-only memory (EEPROM). The device driver does not become available. Hardware support should be contacted.

### *10GbE 4-port Mezzanine PCIe Adapter (a2191007df1033e7)*

The error IDs for the 10GbE 4-port Mezzanine PCIe adapter device driver (a2191007df1033e7) are as follows:

**ERRID_ELXENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_PIO_ERR**
Indicates that the device driver has detected a program IO error. The device driver was unable to troubleshooting the problem. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link

is established again, the device driver logs **ERRID_ELXENT_RCVRY_EXIT**. User intervention is necessary to troubleshooting the problem

**ERRID_ELXENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected

**ERRID_ELXENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_ELXENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_ELXENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_VPD_ERR**
Indicates that the device driver could not able to get VPD (Vital Product Data) from the device, even though device firmware set VPD capabilities.

**ERRID_ELXENT_EEH_ENTRY**
Indicates that the device driver found enhanced error handling (EEH) freeze condition.

**ERRID_ELXENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

**ERRID_ELXENT_MBX_ERR**
Indicates that the device driver got an error when device could not able to complete the request asked by the driver.

**ERRID_ELXENT_EEPROM_ERR**
Indicates that the device driver is in a defined state due to an invalid or bad Electronically erasable programmable read-only memory (EEPROM). The device driver does not become available. Hardware support should be contacted.

### *Int Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203)*

The error IDs for the Int Multifunction Card SR Optical 10GbE (a219100714100904) and Base-TX 10/100/1000 1GbE (a21910071410d203) device driver are as follows:

**ERRID_ELXENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_PIO_ERR**
Indicates that the device driver has detected a program IO error. The device driver was unable to troubleshooting the problem. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_ELXENT_RCVRY_EXIT**. User intervention is necessary to troubleshooting the problem

**ERRID_ELXENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected

**ERRID_ELXENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs **ERRID_ELXENT_RCVRY_EXIT** . User intervention is not necessary for this error unless the problem persists.

**ERRID_ELXENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_VPD_ERR**
Indicates that the device driver could not able to get VPD (Vital Product Data) from the device, even though device firmware set VPD capabilities.

**ERRID_ELXENT_EEH_ENTRY**
Indicates that the device driver found enhanced error handling (EEH) freeze condition.

**ERRID_ELXENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

**ERRID_ELXENT_MBX_ERR**
Indicates that the device driver got an error when device could not able to complete the request asked by the driver.

**ERRID_ELXENT_EEPROM_ERR**
Indicates that the device driver is in a defined state due to an invalid or bad EEPROM. The device driver does not become available. Hardware support should be contacted.

### *Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203)*

The error IDs for the Int Multifunction Card Copper SFP+ 10GbE (a219100714100a04) and Base-TX 10/100/1000 1GbE (a21910071410d203) device driver are as follows:

**ERRID_ELXENT_ADAP_ERR**
Indicates that the adapter failed initialization commands. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_PIO_ERR**
Indicates that the device driver has detected a program IO error. The device driver was unable to troubleshooting the problem. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_LINK_DOWN**
Indicates that the link between the adapter and the network switch is down. The device driver attempts to reestablish the connection after the physical link is reestablished. When the link is established again, the device driver logs **ERRID_ELXENT_RCVRY_EXIT**. User intervention is necessary to troubleshooting the problem

**ERRID_ELXENT_RCVRY_EXIT**
Indicates that a temporary error (such as link down, command error, or transmission error) is corrected

**ERRID_ELXENT_TX_ERR**
Indicates that the device driver has detected a transmission error. The device driver enters an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it logs**ERRID_ELXENT_RCVRY_EXIT**. User intervention is not necessary for this error unless the problem persists.

**ERRID_ELXENT_EEH_SERVICE_ERR**
Indicates that the device driver has detected a error during an attempt to recover from a PCI bus error. User intervention is necessary to troubleshooting the problem.

**ERRID_ELXENT_VPD_ERR**
Indicates that the device driver could not able to get VPD (Vital Product Data) from the device, even though device firmware set VPD capabilities.

**ERRID_ELXENT_EEH_ENTRY**
Indicates that the device driver found enhanced error handling (EEH) freeze condition.

**ERRID_ELXENT_EEH_EXIT**
Indicates that the device driver has successfully recovered from EEH freeze condition.

**ERRID_ELXENT_MBX_ERR**
Indicates that the device driver got an error when device could not able to complete the request asked by the driver.

**ERRID_ELXENT_EEPROM_ERR**
Indicates that the device driver is in a defined state due to an invalid or bad electronically erasable programmable read-only memory (EEPROM). The device driver does not become available. Hardware support should be contacted.

# Graphic Input Devices Subsystem

The graphic input devices subsystem includes the keyboard/sound, mouse, tablet, dials, and lighted programmable-function keys (LPFK) devices.

These devices provide operator input primarily to graphic applications. However, the keyboard can provide system input by means of the console.

The program interface to the input device drivers is described in the **inputdd.h** header file. This header file is available as part of the **bos.adt.graphics** fileset.

## open and close Subroutines

An **open** subroutine call is used to create a channel between the caller and a graphic input device driver.

The keyboard supports two such channels. The most recently created channel is considered the active channel. All other graphic input device drivers support only one channel. The **open** subroutine call is processed normally, except that the *OFLAG* and *MODE* parameters are ignored. The keyboard provides support for the **fp_open** subroutine call; however, only one kernel mode channel can be open at any given time. The **fp_open** subroutine call returns EACCES for all other graphic input devices.

The **close** subroutine is used to remove a channel created by the **open** subroutine call.

## read and write Subroutines

The graphic input device drivers do not support read or write operations.

A read or write to a graphic input device special file behaves as if a read or write was made to **/dev/null**.

## ioctl Subroutines

The ioctl operations provide run-time services.

The special files support the following ioctl operations:

### Keyboard

The ioctl subroutine provides certain keyboard operations that are described in this section.

| Item | Description |
|---|---|
| **IOCINFO** | Returns the **devinfo** structure. |
| **KSQUERYID** | Queries the keyboard device identifier. |
| **KSQUERYSV** | Queries the keyboard service vector. |
| **KSREGRING** | Registers the input ring. |
| **KSRFLUSH** | Flushes the input ring. |
| **KSLED** | Sets and resets the keyboard LEDs. |
| **KSCFGCLICK** | Configures the clicker. |
| **KSVOLUME** | Sets the alarm volume. |

| Item | Description |
|---|---|
| **KSALARM** | Sounds the alarm. |
| **KSTRATE** | Sets the repeat rate. |
| **KSTDELAY** | Sets the delay before repeat. |
| **KSKAP** | Enables and disables the keep-alive poll. |
| **KSKAPACK** | Acknowledges the keep-alive poll. |
| **KSDIAGMODE** | Enables and disables the diagnostics mode. |

**Note:**

1. A non-active channel processes only **IOCINFO**, **KSQUERYID**, **KSQUERYSV**, **KSREGRING**, **KSRFLUSH**, **KSKAP**, and **KSKAPACK**. All other ioctl subroutine calls are ignored without error.
2. The **KSLED**, **KSCFGCLICK**, **KSVOLUME**, **KSALARM**, **KSTRATE**, and **KSTDELAY** ioctl subroutine calls return an **EBUSY** error in the **errno** global variable when the keyboard is in diagnostics mode.
3. The **KSQUERYSV** ioctl subroutine call is only available when the channel is open from kernel mode (with the **fp_open** kernel service).
4. The **KSKAP**, **KSKAPACK**, **KSDIAGMODE** ioctl subroutine calls are only available when the channel is open from user mode.

## Mouse

The ioctl subroutine provides a list of mouse operations described in this section.

| Item | Description |
|---|---|
| **IOCINFO** | Returns the **devinfo** structure. |
| **MQUERYID** | Queries the mouse device identifier. |
| **MREGRING** | Registers the input ring. |
| **MRFLUSH** | Flushes the input ring. |
| **MTHRESHOLD** | Sets the mouse reporting threshold. |
| **MRESOLUTION** | Sets the mouse resolution. |
| **MSCALE** | Sets the mouse scale. |
| **MSAMPLERATE** | Sets the mouse sample rate. |

## Tablet

The tablet ioctl subroutine provides various run-time services that are described in this section.

| Item | Description |
|---|---|
| **IOCINFO** | Returns the **devinfo** structure. |
| **TABQUERYID** | Queries the tablet device identifier. |
| **TABREGRING** | Registers the input ring. |
| **TABFLUSH** | Flushes the input ring. |
| **TABCONVERSION** | Sets the tablet conversion mode. |
| **TABRESOLUTION** | Sets the tablet resolution. |
| **TABORIGIN** | Sets the tablet origin. |
| **TABSAMPLERATE** | Sets the tablet sample rate. |

| Item | Description |
| --- | --- |
| **TABDEADZONE** | Sets the tablet dead zones. |

### GIO (Graphics I/O) Adapter

The ioctl subroutine provides certain GIO adapter run-time operations that are listed in this section.

| Item | Description |
| --- | --- |
| **IOCINFO** | Returns the **devinfo** structure. |
| **GIOQUERYID** | Returns the ID of the attached devices. |

### Dials

The ioctl subroutine provides certain dials operations that are listed in this section.

| Item | Description |
| --- | --- |
| **IOCINFO** | Returns the **devinfo** structure. |
| **DIALREGRING** | Registers the input ring. |
| **DIALRFLUSH** | Flushes the input ring. |
| **DIALSETGRAND** | Sets the dial granularity. |

### LPFK

The ioctl subroutine provides various LPFK operations that are described in this section.

| Item | Description |
| --- | --- |
| **IOCINFO** | Returns the **devinfo** structure. |
| **LPFKREGRING** | Registers the input ring. |
| **LPFKRFLUSH** | Flushes the input ring. |
| **LPFKLIGHT** | Sets and resets the key lights. |

## Input Ring

Data is obtained from graphic input devices by way of a circular First-In First-Out (FIFO) queue or input ring, rather than with a **read** subroutine call.

The memory address of the input ring is registered with an **ioctl** (or **fp_ioctl**) subroutine call. The program that registers the input ring is the owner of the ring and is responsible for allocating, initializing, and freeing the storage associated with the ring. The same input ring can be shared by multiple devices.

The input ring consists of the input ring header followed by the reporting area. The input ring header contains the reporting area size, the head pointer, the tail pointer, the overflow flag, and the notification type flag. Before registering an input ring, the ring owner must ensure that the head and tail pointers contain the starting address of the reporting area. The overflow flag must also be cleared and the size field set equal to the number of bytes in the reporting area. After the input ring has been registered, the owner can modify only the head pointer and the notification type flag.

Data stored on the input ring is structured as one or more event reports. Event reports are placed at the tail of the ring by the graphic input device drivers. Previously queued event reports are taken from the head of the input ring by the owner of the ring. The input ring is empty when the head and tail locations are the same. An overflow condition exists if placement of an event on the input ring would overwrite data that has not been processed. Following an overflow, new event reports are not placed on the input ring until the input ring is flushed via an **ioctl** subroutine or service vector call.

The owner of the input ring is notified when an event is available for processing via a SIGMSG signal or via callback if the channel was created by an **fp_open** subroutine call. The notification type flag in the input ring header specifies whether the owner should be notified each tine an event is placed on the ring or only when an event is placed on an empty ring.

## Management of Multiple Keyboard Input Rings

When multiple keyboard channels are opened, keyboard events are placed on the input ring associated with the most recently opened channel.

When this channel is closed, the alternate channel is activated and keyboard events are placed on the input ring associated with that channel.

## Event Report Formats

Each event report consists of an identifier followed by the report size in bytes, a time stamp (system time in milliseconds), and one or more bytes of device-dependent data. The value of the identifier is specified when the input ring is registered.

The program requesting the input-ring registration is responsible for identifier uniqueness within the input-ring scope.

**Note:** Event report structures are placed on the input-ring without spacing. Data wraps from the end to the beginning of the reporting area. A report can be split on any byte boundary into two non-contiguous sections.

The event reports are as follows:

### *Keyboard*

The keyboard event report format contains the following fields.

| Item | Description |
|---|---|
| ID | Specifies the report identifier. |
| Length | Specifies the report length. |
| Time stamp | Specifies the system time (in milliseconds). |
| Key position code | Specifies the key position code. |
| Key scan code | Specifies the key scan code. |
| Status flags | Specifies the status flags. |

### *Tablet*

The tablet event report contains the following identifiers.

| Item | Description |
|---|---|
| ID | Specifies the report identifier. |
| Length | Specifies the report length. |
| Time stamp | Specifies the system time (in milliseconds). |
| Absolute X | Specifies the absolute *X* coordinate. |
| Absolute Y | Specifies the absolute *Y* coordinate. |

### *LPFK*

Following are the identifiers for LPFK event report:

| Item | Description |
|---|---|
| ID | Specifies the report identifier. |

| Item | Description |
|---|---|
| Length | Specifies the report length. |
| Time stamp | Specifies the system time (in milliseconds). |
| Number of key pressed | Specifies the number of the key pressed. |

### *Dials*

The dial event reports are as follows:

| Item | Description |
|---|---|
| ID | Specifies the report identifier. |
| Length | Specifies the report length. |
| Time stamp | Specifies the system time (in milliseconds). |
| Number of dial changed | Specifies the number of the dial changed. |
| Delta change | Specifies delta dial rotation. |

### *Mouse (Standard Format)*

The mouse event report in standard format contains the following identifiers:

| Item | Description |
|---|---|
| ID | Specifies the report identifier. |
| Length | Specifies the report length. |
| Time stamp | Specifies the system time (in milliseconds). |
| Delta X | Specifies the delta mouse motion along the *X* axis. |
| Delta Y | Specifies the delta mouse motion along the *Y* axis. |
| Button status | Specifies the button status. |

## Mouse (Extended Format)

The mouse event report in extended format contains the following fields:

| Item | Description |
|---|---|
| ID | Specifies the report identifier. |
| Length | Specifies the report length. |
| Time stamp | Specifies the system time (in milliseconds). |
| Format | Specifies the format of additional fields.<br>**Format 1:**<br>• **Status:** Specifies the extended button status.<br>• **Delta Wheel:** Specifies the delta wheel movement.<br>**Format 2:**<br>• **Button Status:** Specifies the button status.<br>• **Delta X:** Specifies the delta mouse motion along the *X* axis.<br>• **Delta Y:** Specifies the delta mouse motion along the *Y* axis.<br>• **Delta Wheel:** Specifies the delta wheel movement. |

## Keyboard Service Vector

The keyboard service vector provides a limited set of keyboard-related and sound-related services for kernel extensions.

The following services are available:

- Sound alarm
- Enable and disable secure attention key (SAK)
- Flush input queue

The address of the service vector is obtained with the fp_ioctl subroutine call during a non-critical period. The kernel extension can later invoke the service using an indirect call as follows:

**(***_ServiceVector_**[**_ServiceNumber_**]) (dev_t** _DeviceNumber_**, caddr_t** _Arg_**);**

where:

- The service vector is a pointer to the service vector obtained by the **KSQUERYSV** fp_loctl subroutine call.
- The *ServiceNumber* parameter is defined in the **inputdd.h** file.
- The *DeviceNumber* parameter specifies the major and minor numbers of the keyboard.
- The *Arg* parameter points to a **ksalarm** structure for alarm requests and a **uint** variable for SAK enable and disable requests. The *Arg* parameter is NULL for flush queue requests.

If successful, the function returns a value of 0 is returned. Otherwise, the function returns an error number defined in the **errno.h** file. Flush-queue and enable/disable-SAK requests are always processed, but alarm requests are ignored if the kernel extension's channel is inactive.

The following example uses the service vector to sound the alarm:

```
/* pinned data structures                                */
/* This example assumes that pinning is done elsewhere. */
int (**ksvtbl) ();
struct ksalarm alarm;
dev_t devno;

/* get address of service vector                */
/* This should be done in a noncritical section */
if (fp_ioctl(fp, KSQUERYSV, &ksvtbl, 0)) {
 /* error recovery */
}
.
.
.

/* critical section                              */
/* sound alarm for 1 second using service vector */
alarm.duration = 128;
alarm.frequency = 100;

if ((*ksvtbl[KSVALARM]) (devno, &alarm)) {
 /* error recovery */
}
```

## Special Keyboard Sequences

Special keyboard sequences are provided for the Secure Attention Key (SAK) and the Keep Alive Poll (KAP).

### *Secure Attention Key*

The user requests a secure shell by keying a secure attention. The keyboard driver interprets the key sequence CTRL x r as the SAK. An indirect call using the keyboard service vector enables and disables the detection of this key sequence.

If detection of the SAK is enabled, a SAK causes the SAK callback to be invoked. The SAK callback is invoked even if the input ring is inactive due to a user process issuing an open to the keyboard special file. The SAK callback runs within the interrupt environment.

### *Keep Alive Poll*

The keyboard device driver supports a special key sequence that kills the process that owns the keyboard.

This sequence must first be defined with a **KSKAP** ioctl operation. After this sequence is defined, the keyboard device driver sends a **SIGKAP** signal to the process that owns the keyboard when the special sequence is entered on the keyboard. The process that owns the keyboard must acknowledge the **KSKAP** signal with a **KSKAPACK** ioctl within 30 seconds or the keyboard driver will terminate the process with a **SIGKILL** signal. The KAP is enabled on a per-channel basis and is unavailable if the channel is owned by a kernel extension.

# Low Function Terminal Subsystem

The low function terminal (lft) interface is a pseudo-device driver that interfaces with device drivers for the system keyboard and display adapters. The lft interface adheres to all standard requirements for pseudo-device drivers and has all the entry points and configuration code as required by the device driver architecture. This section gives a high-level description of the various configuration methods and entry points provided by the lft interface.

All the device drivers controlled by the lft interface are also used by AIXwindows. Consequently, along with the functions required for the tty subsystem interface, the lft interface provides the functions required by AIXwindows interfaces with display device driver adapters.

**Related information**

National Language Support Overview
Setting National Language Support for Devices
Locales
Keyboard Overview
Understanding the Japanese Input Method (JIM)
Understanding the Korean Input Method (KIM)
Understanding the Traditional Chinese Input Method (TIM)

## Low Function Terminal Interface Functional Description

The functional description of lft is explained in terms of its configuration, terminal emulation, ioctl support for AIXwindows, lft to keyboard interface, and display device driver interface and entry points.

### Configuration

The lft interface uses the common define, undefine, and unconfiguration methods standard for most devices.

**Note:** The lft interface does not support any change method for dynamically changing the lft configuration. Instead, use the **-P** flag with the **chdev** command. The changes become effective the next time the lft interface is configured.

The configuration process for the lft opens all display device drivers. To define the default display and console, select the default display and console during the console configuration process. If a graphics display is chosen as the system console, it automatically becomes the default display. The lft interface displays text on the default display.

The configuration process for the lft interface queries the ODM database for the available fonts and software keyboard map for the current session.

### Terminal Emulation

The lft interface is a stream-based tty subsystem. The lft interface provides VT100 (or IBM® 3151) terminal emulation for the standard part of the ANSI 3.64 data stream. All line discipline handling is performed in the layers above the lft interface. The lft interface does not support virtual terminals.

The lft interface supports multiple fonts to handle the different screen sizes and resolutions necessary in providing a 25x80 character display on various display adapters.

**Note:** Applications requiring hft extensions need to use aixterm.

### IOCTLS Needed for AIXwindows Support

AIXwindows and the lft interface share the system keyboard and display device drivers.

To prevent screen and keyboard inconsistencies, a set of ioctl coordinates usage between AIXwindows and the lft interface. On a system with multiple displays, the lft interface can still use the default display as long as AIXwindows is using another display.

**Note:** The lft interface provides ioctl support to set and change the default display.

### Low Function Terminal to System Keyboard Interface

The lft interface with the system keyboard uses an input ring mechanism.

The details of the keyboard driver ioctls, as well as the format and description of this input ring, are provided in “Graphic Input Devices Subsystem” on page 215. The keyboard device driver passes raw keystrokes to the lft interface. These keystrokes are converted to the appropriate code point using keyboard tables. The use of keyboard-language-dependent keyboard tables ensures that the lft interface provides National Language Support.

### Low Function Terminal to Display Device Driver Interface

The lft uses a device independent interface known as the virtual display driver (vdd) interface. Because the lft interface has no virtual terminal or monitor mode support, some of the vdd entry points are not used by the lft.

The display drivers might enqueue font request through the font process started during lft initialization. The font process pins and unpins the requested fonts for **DMA** to the display adapter.

### Low Function Terminal Device Driver Entry Points

The lft interface supports the open, close, read, write, ioctl, and configuration entry points.

## Components Affected by the Low Function Terminal Interface

All the device drivers controlled by the lft interface are also used by AIXwindows. Consequently, the lft interface affects other components as well.

### Configuration User Commands

The lft interface is a pseudo-device driver. Consequently, the system configuration process does not detect the lft interface as it does an adapter. The system provides for pseudo-device drivers to be started through **Config_Rules**.

To start the lft interface, use the **startlft** program.

Supported commands include:

- **lsfont**

- **mkfont**
- **chfont**
- **lskbd**
- **chkbd**
- **lsdisp** (see note)
- **chdisp** (see note)

**Note:**

1. *lsdisp* outputs the logical device name instead of the instance number.
2. *chdisp* uses the ioctl interface to the lft to set the requested display.

## Display Device Driver

A display device driver is required for each supported display adapter.

The display device drivers provide all the standard interfaces (such as config, initialize, terminate, and so forth) required in any AIX® 4.1 (or later) device drivers. The only device switch table entries supported are open, close, config, and ioctl. All other device switch table entries are set to nodev. In addition, the display device drivers provide a set of ioctls for use by AIXwindows and diagnostics to perform device specific functions such as get bus access, bus memory address, DMA operations, and so forth.

## Rendering Context Manager

The Rendering Context Manager (RCM) is a loadable module.

**Note:** Previously, the high functional terminal interface provided AIXwindows with the **gsc_handle**. This handle is used in all of the **aixgsc** system calls. The RCM provides this service for the lft interface.

To ensure that lft can recover the display in case AIXwindows should terminate abnormally, AIXwindows issues the ioctl to RCM after opening the pseudo-device. RCM passes on the ioctl to the lft. This way, the close function in RCM is invoked (Because AIXwindows is the only application that has opened RCM), and RCM notifies the lft interface to start reusing the display. To support this communication, the RCM provides the required ioctl support.

### *The RCM to lft Interface Initialization*

The RCM to lft interface initialization includes the following:

1. RCM performs the open **/dev/lft**.
2. Upon receiving a list of displays from X, RCM passes the information to the lft through an ioctl.
3. RCM resets the adapter.

### *If AIXwindows Terminates Abnormally*

If AIXwindows terminates abnormally, the following might happen:

1. RCM receives notification from X about the displays it was using.
2. RCM resets the adapter.
3. RCM passes the information to the lft by way of an ioctl.

### *AIXwindows to lft Initialization*

The AIXwindows to lft initialization includes the following:

1. AIXwindows opens **/dev/rcm**.
2. AIXwindows gets the **gsc_handle** from RCM via an ioctl.
3. AIXwindows becomes a graphics process aixgsc (MAKE_GP, ...)
4. AIXwindows, through an ioctl, informs RCM about the displays it wishes to use.
5. AIXwindows opens all of the input devices it needs and passes the same input ring to each of them.

### *Upon Normal Termination*

On normal termination, the following actions are performed.

1. X issues a close to all of the input devices it opened.
2. X informs RCM, through an ioctl, about the displays it was using.

## Diagnostics

Diagnostics and other applications that require access to the graphics adapter use the AIXwindows to lft interface.

# Accented Characters

Here are the valid sets of characters for each of the diacritics that the Low Function Terminal (LFT) subsystem uses to validate the two-key nonspacing character sequence.

The following section provides a list of diacritics supported by the HFT LFT subsystem:

## List of Diacritics Supported by the HFT LFT Subsystem

There are seven diacritic characters for which sets of characters are provided.

- Acute
- Grave
- Circumflex
- Umlaut
- Tilde
- Overcircle
- Cedilla

### *Valid Sets of Characters (Categorized by Diacritics)*

This section lists the code values for the seven diacritic characters.

| **Item** | **Description** |
|---|---|
| **Acute Function** | **Code Value** |
| Acute accent | 0xef |
| Apostrophe (acute) | 0x27 |
| e Acute small | 0x82 |
| e Acute capital | 0x90 |
| a Acute small | 0xa0 |
| i Acute small | 0xa1 |
| o Acute small | 0xa2 |
| u Acute small | 0xa3 |
| a Acute capital | 0xb5 |
| i Acute capital | 0xd6 |
| y Acute small | 0xec |
| y Acute capital | 0xed |
| o Acute capital | 0xe0 |
| u Acute capital | 0xe9 |

The following are grave function code values:

| Item | Description |
|---|---|
| **Grave Function** | **Code Value** |
| Grave accent | 0x60 |
| a Grave small | 0x85 |
| e Grave small | 0x8a |
| i Grave small | 0x8d |
| o Grave small | 0x95 |
| u Grave small | 0x97 |
| a Grave capital | 0xb7 |
| e Grave capital | 0xd4 |
| i Grave capital | 0xde |
| o Grave capital | 0xe3 |
| u Grave capital | 0xeb |

The following are circumflex function code values:

| Item | Description |
|---|---|
| **Circumflex Function** | **Code Value** |
| **^** Circumflex accent | 0x5e |
| a Circumflex small | 0x83 |
| e Circumflex small | 0x88 |
| i Circumflex small | 0x8c |
| o Circumflex small | 0x93 |
| u Circumflex small | 0x96 |
| a Circumflex capital | 0xb6 |
| e Circumflex capital | 0xd2 |
| i Circumflex capital | 0xd7 |
| o Circumflex capital | 0xe2 |
| u Circumflex capital | 0xea |

The following are umlaut function code values:

| Item | Description |
|---|---|
| **Umlaut Function** | **Code Value** |
| Umlaut accent | 0xf9 |
| u Umlaut small | 0x81 |
| a Umlaut small | 0x84 |
| e Umlaut small | 0x89 |
| i Umlaut small | 0x8b |
| a Umlaut capital | 0x8e |

| Item | Description |
|---|---|
| O Umlaut capital | 0x99 |
| u Umlaut capital | 0x9a |
| e Umlaut capital | 0xd3 |
| i Umlaut capital | 0xd8 |

The following are tilde function code values:

| Item | Description |
|---|---|
| **Tilde Function** | **Code Value** |
| Tilde accent | 0x7e |
| n Tilde small | 0xa4 |
| n Tilde capital | 0xa5 |
| a Tilde small | 0xc6 |
| a Tilde capital | 0xc7 |
| o Tilde small | 0xe4 |
| o Tilde capital | 0xe5 |
| Overcircle Function | Code Value |
| Overcircle accent | 0x7d |
| a Overcircle small | 0x86 |
| a Overcircle capital | 0x8f |
| Cedilla Function | Code Value |
| Cedilla accent | 0xf7 |
| c Cedilla capital | 0x80 |
| c Cedilla small | 0x87 |

# Logical Volume Subsystem

A logical volume subsystem provides flexible access and control for complex physical storage systems.

Review the following information before you proceed:

- **readx** subroutine
- **write** subroutine
- **lvdd** special file
- **buf** structure
- **bread** kernel service
- **bwrite** kernel service
- **iodone** kernel service

## Direct Access Storage Devices (DASDs)

Direct access storage devices (DASDs) are *fixed* or *removable* storage devices. Typically, these devices are rotating disk drives or solid state disks. A fixed storage device is any storage device defined during system configuration to be an integral part of the system DASD. The operating system detects an error if a fixed storage device is not available at some time during normal operation.

A removable storage device is any storage device defined by the person who administers your system during system configuration to be an optional part of the system DASD. The removable storage device can be removed from the system at any time during normal operation. As long as the device is logically unmounted first, the operating system does not detect an error.

The following types of devices are not considered DASD and are not supported by the logical volume manager (LVM):

- CD-ROM (compact disk read-only memory)
- DVD
- WORM (write-once read-many)

## Physical volumes

A logical volume is a portion of a physical volume viewed by the system as a volume. Logical records are records defined in terms of the information they contain rather than physical attributes.

A physical volume is a DASD structured for requests at the physical level, that is, the level at which a processing unit can request device-independent operations on a physical block address basis. A physical volume is composed of the following:

- A device-dependent reserved area
- A variable number of physical blocks that serve as DASD descriptors
- An integral number of partitions, each containing a fixed number of physical blocks

When performing I/O at a physical level, no bad-block relocation is supported. Bad blocks are not hidden at this level as they are at the logical level. Typical operations at the physical level are `read-physical-block` and `write-physical-block`. For more information on bad blocks, see "Understanding Logical Volumes and Bad Blocks" on page 233.

The following are terms used when discussing DASD volumes:

| Item | Description |
|---|---|
| **block** | A contiguous, 512 byte or 4096 byte region of a physical volume that corresponds in size to a DASD sector |
| **partition** | A set of blocks (with sequential cylinder, head, and sector numbers) contained within a single physical volume |

The number of blocks in a partition, as well as the number of partitions in a given physical volume, are fixed when the physical volume is installed in a volume group. Every physical volume in a volume group has exactly the same partition size. There is no restriction on the types of DASDs (for example, Serial Attached SCSI (SAS), Fibre Channel, iSCSI, virtual SCSI, or Non-volatile memory express (NVMe) that can be placed in a given volume group.

**Note:** A given physical volume must be assigned to a volume group before that physical volume can be used by the LVM.

### Physical Volume Implementation Limitations

When composing a physical volume from a DASD, the following implementation restrictions apply to DASD characteristics:

- 1 to 32 physical volumes per volume group
- 1 to 128 physical volumes in a big volume group
- The partition size is restricted to 2**$n$ bytes, for 20 <= $n$ <= 30
- The physical block size is restricted to 512 or 4096 bytes

## Physical Volume Layout

A physical volume consists of a logically contiguous string of physical sectors.

Sectors are numbered 0 through the last physical sector number (LPSN) on the physical volume. The total number of physical sectors on a physical volume is *LPSN* + 1. The actual physical location and physical order of the sectors are transparent to the sector numbering scheme.

**Note:** Sector numbering applies to user-accessible data sectors only. Spare sectors and Customer-Engineer (CE) sectors are not included. CE sectors are reserved for use by diagnostic test routines or microcode.

## Reserved Sectors on a Physical Volume

A physical volume reserves the first 128 sectors to store various types of DASD configuration and operation information. The **/usr/include/sys/hd_psn.h** file describes the information stored on the reserved sectors. The locations of the items in the reserved area are expressed as physical sector numbers in this file, and the lengths of those items are in number of sectors.

The 128-sector reserved area of a physical volume includes a boot record, the bad-block directory, the LVM record, and the mirror write consistency (MWC) record. The boot record consists of one sector containing information that allows the read-only system (ROS) to boot the system. A description of the boot record can be found in the **/usr/include/sys/bootrecord.h** file.

The boot record also contains the `pv_id` field. This field is a 64-bit number uniquely identifying a physical volume. This identifier might be assigned by the manufacturer of the physical volume. However, if a physical volume is part of a volume group, the `pv_id` field will be assigned by the LVM.

The bad-block directory records the blocks on the physical volume that have been diagnosed as unusable. The structure of the bad-block directory and its entries can be found in the **/usr/include/sys/bbdir.h** file.

The LVM record consists of one sector and contains information used by the LVM when the physical volume is a member of the volume group. The LVM record is described in the **/usr/include/lvmrec.h** file.

The MWC record consists of one sector. It identifies which logical partitions might be inconsistent if the system is not shut down properly. When the volume group is varied back online for use, this information is used to make logical partitions consistent again.

## Sectors Reserved for the Logical Volume Manager (LVM)

If a physical volume is part of a volume group, the physical volume is used by the LVM and contains two additional reserved areas. One area contains the volume group descriptor area/volume group status area and follows the first 128 reserved sectors. The other area is at the end of the physical volume reserved as a relocation pool for bad blocks that must be software-relocated. Both of these areas are described by the LVM record. The space between these last two reserved areas is divided into equal-sized partitions.

The volume group descriptor area (VGDA) is divided into the following:

- The volume group header. This header contains general information about the volume group and a time stamp used to verify the consistency of the VGDA.
- A list of logical volume entries. The logical volume entries describe the states and policies of logical volumes. This list defines the maximum number of logical volumes allowed in the volume group. The maximum is specified when a volume group is created.
- A list of physical volume entries. The size of the physical volume list is variable because the number of entries in the partition map can vary for each physical volume. For example, a 200 MB physical volume with a partition size of 1 MB has 200 partition map entries.
- A name list. This list contains the special file names of each logical volume in the volume group.
- A volume group trailer. This trailer contains an ending time stamp for the volume group descriptor area.

When a volume group is varied online, a majority (also called a quorum) of VGDAs must be present to perform recovery operations unless you have specified the **force** flag. (The `vary-on` operation, performed by using the **varyonvg** command, makes a volume group available to the system).

**Attention:** Use of the **force** flag can result in data inconsistency.

A volume group with only one physical volume must contain two copies of the physical volume group descriptor area. For any volume group containing more than one physical volume, there are at least three on-disk copies of the volume group descriptor area. The default placement of these areas on the physical volume is as follows:

- For the first physical volume installed in a volume group, two copies of the volume group descriptor area are placed on the physical volume.
- For the second physical volume installed in a volume group, one copy of the volume group descriptor area is placed on the physical volume.
- For the third physical volume installed in a volume group, one copy of the volume group descriptor area is placed on the physical volume. The second copy is removed from the first volume.
- For additional physical volumes installed in a volume group, one copy of the volume group descriptor area is placed on the physical volume.

When a `vary-on` operation is performed, a majority of copies of the volume group descriptor area must be able to come online before the `vary-on` operation is considered successful. A quorum ensures that at least one copy of the volume group descriptor areas available to perform recovery was also one of the volume group descriptor areas that were online during the previous `vary-off` operation. If not, the consistency of the volume group descriptor area cannot be ensured.

The volume group status area (VGSA) contains the status of all physical volumes in the volume group. This status is limited to active or missing. The VGSA also contains the state of all allocated physical partitions (PP) on all physical volumes in the volume group. This state is limited to active or stale. A PP with a stale state is not used to satisfy a read request and is not updated on a write request.

A PP changes from stale to active after a successful resynchronization of the logical partition (LP) that has multiple copies, or mirrors, and is no longer consistent with its peers in the LP. This inconsistency can be caused by a write error or by not having a physical volume available when the LP is written to or updated.

A PP changes from stale to active after a successful resynchronization of the LP. A resynchronization operation issues resynchronization requests starting at the beginning of the LP and proceeding sequentially through its end. The LVDD reads from an active partition in the LP and then writes that data to any stale partition in the LP. When the entire LP has been traversed, the partition state is changed from stale to active.

Normal I/O can occur concurrently in an LP that is being resynchronized.

**Note:** If a write error occurs in a stale partition while a resynchronization is in progress, that partition remains stale.

If all stale partitions in an LP encounter write errors, the resynchronization operation is ended for this LP and must be restarted from the beginning.

The `vary-on` operation uses the information in the VGSA to initialize the LVDD data structures when the volume group is brought online.

## Understanding the Logical Volume Device Driver

The Logical Volume Device Driver (LVDD) is a pseudo-device driver that operates on logical volumes through the **/dev/lv***n* special file.

Like the physical disk device driver, this pseudo-device driver provides character and block entry points with compatible arguments. Each volume group has an entry in the kernel device switch table. Each entry contains entry points for the device driver and a pointer to the volume group data structure. The logical volumes of a volume group are distinguished by their minor numbers.

**Attention:** Each logical volume has a control block located in the first 512 bytes. Data begins in the second 512-byte block. Care must be taken when reading and writing directly to the logical volume, such as when using applications that write to raw logical volumes, because the control

block is not protected from such writes. If the control block is overwritten, commands that use the control block will use default information instead.

Character I/O requests are performed by issuing a read or write request on a **/dev/rlv***n* character special file for a logical volume. The read or write is processed by the file system SVC handler, which calls the LVDD **ddread** or **ddwrite** entry point. The **ddread** or **ddwrite** entry point transforms the character request into a block request. This is done by building a buffer for the request and calling the LVDD **ddstrategy** entry point.

Block I/O requests are performed by issuing a read or write on a block special file **/dev/lv***n* for a logical volume. These requests go through the SVC handler to the **bread** or **bwrite** block I/O kernel services. These services build buffers for the request and call the LVDD **ddstrategy** entry point. The LVDD **ddstrategy** entry point then translates the logical address to a physical address (handling bad block relocation and mirroring) and calls the appropriate physical disk device driver.

On completion of the I/O, the physical disk device driver calls the **iodone** kernel service on the device interrupt level. This service then calls the LVDD I/O completion-handling routine. Once this is completed, the LVDD calls the **iodone** service again to notify the requester that the I/O is completed.

The LVDD is logically split into top and bottom halves. The top half contains the **ddopen**, **ddclose**, **ddread**, **ddwrite**, **ddioctl**, and **ddconfig** entry points. The bottom half contains the **ddstrategy** entry point, which contains block read and write code. This is done to isolate the code that must run fully pinned and has no access to user process context. The bottom half of the device driver runs on interrupt levels and is not permitted to page fault. The top half runs in the context of a process address space and can page fault.

## Data Structures

The interface to the **ddstrategy** entry point is one or more logical **buf** structures in a list.

The logical **buf** structure is defined in the **/usr/include/sys/buf.h** file and contains all needed information about an I/O request, including a pointer to the data buffer. The **ddstrategy** entry point associates one or more (if mirrored) physical **buf** structures (or **pbufs**) with each logical **buf** structure and passes them to the appropriate physical device driver.

The **pbuf** structure is a standard **buf** structure with some additional fields. The LVDD uses these fields to track the status of the physical requests that correspond to each logical I/O request. A pool of pinned **pbuf** structures is allocated and managed by the LVDD.

There is one device switch entry for each volume group defined on the system. Each volume group entry contains a pointer to the volume group data structure describing it.

## Top Half of LVDD

The top half of the LVDD contains the code that runs in the context of a process address space and can page fault.

It contains the following entry points:

| Item | Description |
|---|---|
| **ddopen** | Called by the file system when a logical volume is mounted, to open the logical volume specified. |
| **ddclose** | Called by the file system when a logical volume is unmounted, to close the logical volume specified. |
| **ddconfig** | Initializes data structures for the LVDD. |

| Item | Description |
|---|---|
| **ddread** | Called by the **read** subroutine to translate character I/O requests to block I/O requests. This entry point verifies that the request is on a 512-byte boundary and is a multiple of 512 bytes in length.<br><br>Most of the time a request will be sent down as a single request to the LVDD **ddstrategy** entry point which handles logical block I/O requests. However, the **ddread** routine might divide very large requests into multiple requests that are passed to the LVDD **ddstrategy** entry point.<br><br>If the *ext* parameter is set (called by the **readx** subroutine), the **ddread** entry point passes this parameter to the LVDD **ddstrategy** routine in the `b_options` field of the buffer header. |
| **ddwrite** | Called by the **write** subroutine to translate character I/O requests to block I/O requests. The LVDD **ddwrite** routine performs the same processing for a write request as the LVDD **ddread** routine does for read requests. |
| **ddioctl** | Supports the following operations:<br><br>**CACLNUP**<br>Causes the mirror write consistency (MWC) cache to be written to all physical volumes (PVs) in a volume group.<br><br>**IOCINFO, XLATE**<br>Return LVM configuration information and PP status information.<br><br>**LV_INFO**<br>Provides information about a logical volume.<br><br>**PBUFCNT**<br>Increases the number of physical buffer headers (pbufs) in the LVM pbuf pool. |

## Bottom Half of the LVDD

The bottom half of the device driver supports the **ddstrategy** entry point. This entry point processes all logical block requests and performs the following functions:

- Validates I/O requests.
- Checks requests for conflicts (such as overlapping block ranges) with requests currently in progress.
- Translates logical addresses to physical addresses.
- Handles mirroring and bad-block relocation.

The bottom half of the LVDD runs on interrupt levels and, as a result, is not permitted to page fault.

Each logical I/O request passes down through the bottom three layers before reaching the physical disk device driver. Once the I/O is complete, the request returns back up through the layers to handle the I/O completion processing at each layer. Finally, control returns to the original requestor.

The bottom half of the LVDD is divided into the following three layers:

### *Strategy Layer*

The strategy layer deals only with logical requests.

The **ddstrategy** entry point is called with one or more logical **buf** structures. A list of **buf** structures for requests that are not blocked are passed to the second layer, the scheduler.

### *Scheduler Layer*

The scheduler layer schedules physical requests for logical operations and handles mirroring and the MWC cache. For each logical request the scheduler layer schedules one or more physical requests. These requests involve translating logical addresses to physical addresses, handling mirroring, and calling the LVDD physical layer with a list of physical requests.

When a physical I/O operation is complete for one phase or mirror of a logical request, the scheduler initiates the next phase (if there is one). If no more I/O operations are required for the request, the scheduler calls the strategy termination routine. This routine notifies the originator that the request has been completed.

The scheduler also handles the MWC cache for the volume group. If a logical volume is using mirror write consistency, then requests for this logical volume are held within the scheduling layer until the MWC cache blocks can be updated on the target physical volumes. When the MWC cache blocks have been updated, the request proceeds with the physical data write operations.

When MWC is being used, system performance can be adversely affected. This is caused by the overhead of logging or journalling that a write request is active in one or more logical track groups (LTGs) (128K, 256K, 512K or 1024K). This overhead is for mirrored writes only. It is necessary to guarantee data consistency between mirrors particularly if the system crashes before the write to all mirrors has been completed.

Mirror write consistency can be turned off for an entire logical volume. It can also be inhibited on a request basis by turning on the **NO_MWC** flag as defined in the **/usr/include/sys/lvdd.h** file.

### *Physical Layer*

The physical layer of the LVDD handles startup and termination of the physical request. The physical layer calls a physical disk device driver's **ddstrategy** entry point with a list of **buf** structures linked together. In turn, the physical layer is called by the **iodone** kernel service when each physical request is completed.

This layer also performs bad-block relocation and detection/correction of bad blocks, when necessary. These details are hidden from the other two layers.

## Interface to Physical Disk Device Drivers

Physical disk device drivers adhere to the criteria that are listed in this section if they are to be accessed by the LVDD.

- Disk block size must be 512 bytes.
- The physical disk device driver needs to accept a list of requests defined by **buf** structures, which are linked together by the `av_forw` field in each **buf** structure.
- For unrecoverable media errors, physical disk device drivers need to set the following:
  - The **B_ERROR** flag must be set to on (defined in the **/usr/include/sys/buf.h** file) in the `b_flags` field.
  - The `b_error` field must be set to **E_MEDIA** (defined in the **/usr/include/sys/errno.h** file).
  - The `b_resid` field must be set to the number of bytes in the request that were not read or written successfully. The `b_resid` field is used to determine the block in error.

    **Note:** For write requests, the LVDD attempts to hardware-relocate the bad block. If this is unsuccessful, then the block is software-relocated. For read requests, the information is recorded and the block is relocated on the next write request to that block.
- For a successful request that generated an excessive number of retries, the device driver can return good data. To indicate this situation it must set the following:
  - The `b_error` field is set to **ESOFT**; this is defined in the **/usr/include/sys/errno.h** file.
  - The `b_flags` field has the **B_ERROR** flag set to on.
  - The `b_resid` field is set to a count indicating the first block in the request that had excessive retries. This block is then relocated.
- The physical disk device driver needs to accept a request of one block with **HWRELOC** (defined in the **/usr/include/sys/lvdd.h** file) set to on in the `b_options` field. This indicates that the device driver is to perform a hardware relocation on this request. If the device driver does not support hardware relocation the following should be set:
  - The `b_error` field is set to **EIO**; this is defined in the **/usr/include/sys/errno.h** file.

  - The `b_flags` field has the **B_ERROR** flag set on.
  - The `b_resid` field is set to a count indicating the first block in the request that has excessive retries.
- The physical disk device driver should support the system dump interface as defined.
- The physical disk device driver must support write verification on an I/O request. Requests for write verification are made by setting the `b_options` field to **WRITEV**. This value is defined in the **/usr/include/sys/lvdd.h** file.

# Understanding Logical Volumes and Bad Blocks

The physical layer of the logical volume device driver (LVDD initiates all bad-block processing.

The physical layer then isolates all of the decision making from the physical disk device driver. This happens so the physical disk device driver does not need to handle mirroring, which is the duplication of data transparent to the user.

## Relocating Bad Blocks

The physical layer of the LVDD checks each physical request to see if there are any known software-relocated bad blocks in the request. The LVDD determines if a request contains known software-relocated bad blocks by hashing the physical address. Then a hash chain of the LVDD defects directory is searched to see if any bad-block entries are in the address range of the request.

If bad blocks exist in a physical request, the request is split into pieces. The first piece contains any blocks up to the relocated block. The second piece contains the relocated block (the relocated address is specified in the bad-block entry) of the defects directory. The third piece contains any blocks after the relocated block to the end of the request or to the next relocated block. These separate pieces are processed sequentially until the entire request has been satisfied.

Once the I/O for the first of the separate pieces has completed, the **iodone** kernel service calls the LVDD physical layer's termination routine (specified in the `b_done` field of the **buf** structure). The termination routine initiates I/O for the second piece of the original request (containing the relocated block), and then for the third piece. When the entire physical operation is completed, the appropriate scheduler's policy routine (in the second layer of the LVDD) is called to start the next phase of the logical operation.

## Detecting and Correcting Bad Blocks

If a logical volume is mirrored, a newly detected bad block is fixed by relocating that block. A good mirror is read and then the block is relocated using data from the good mirror. With mirroring, the user does not need to know when bad blocks are found. However, the physical disk device driver does log permanent I/O errors so the user can determine the rate of media surface errors.

When a bad block is detected during I/O, the physical disk device driver sets the error fields in the **buf** structure to indicate that there was a media surface error. The physical layer of the LVDD then initiates any bad-block processing that must be done.

If the operation was a nonmirrored read, the block is not relocated because the data in the relocated block is not initialized until a write is performed to the block. To support this delayed relocation, an entry for the bad block is put into the LVDD defects directory and into the bad-block directory on disk. These entries contain no relocated block address and the status for the block is set to indicate that relocation is desired.

On each I/O request, the physical layer checks whether there are any bad blocks in the request. If the request is a write and contains a block that is in a `relocation-desired` state, the request is sent to the physical disk device driver with safe hardware relocation requested. If the request is a read, a read of the known defective block is attempted.

If the operation was a read operation in a mirrored LP, a request to read one of the other mirrors is initiated. If the second read is successful, then the read is turned into a write request and the physical disk device driver is called with safe hardware relocation specified to fix the bad mirror.

If the hardware relocation fails or the device does not support safe hardware relocation, the physical layer of the LVDD attempts software relocation. At the end of each volume is a reserved area used by the LVDD as a pool of relocation blocks. When a bad block is detected and the disk device driver is unable to relocate the block, the LVDD picks the next unused block in the relocation pool and writes to this new location. A new entry is added to the LVDD defects directory in memory (and to the bad-block directory on disk) that maps the bad-block address to the new relocation block address. Any subsequent I/O requests to the bad-block address are routed to the relocation address.

**Attention:** Formatting a fixed disk deletes any data on the disk. Format a fixed disk only when absolutely necessary and preferably after backing up all data on the disk.

If you need to format a fixed disk completely (including reinitializing any bad blocks), use the formatting function supplied by the **diag** command. (The **diag** command typically, but not necessarily, writes over all data on a fixed disk. Refer to the documentation that comes with the fixed disk to determine the effect of formatting with the **diag** command.)

# Printer Addition Management Subsystem

If you are configuring a printer for your system, there are basically two types of printers: printers already supported by the operating system and new printer types.

Review the following information for more information about printers:

- Printer Support
- **mkvirprt** command
- **piopredef** command
- **piocmdout**, **piogetstr**, and **piogetvals** subroutines

## Printer Types Currently Supported

Use the **mkvirprt** command to create a customized printer file for your printer.

To configure a supported type of printer, you need only to run the **mkvirprt** command. This customized printer file, which is in the **/var/spool/lpd/pio/@local/custom** directory, describes the specific parameters for your printer. For more information see Configuring a Printer without Adding a Queue in *Printers and printing*.

## Printer Types Currently Unsupported

To configure a currently unsupported type of printer, you must develop and add a predefined printer definition for your printer. This new option is then entered in the list of available choices when the user selects a printer to configure for the system. The actual data used by the printer subsystem comes from the Customized printer definition created by the **mkvirprt** command.

"Adding a New Printer Type to Your System" on page 235 provides general instructions for adding an undefined printer. To add an undefined printer, you modify an existing printer definition. Undefined printers fall into two categories:

- Printers that closely emulate a supported printer. You can use SMIT or the virtual printer commands to make the changes you need.
- Printers that do not emulate a supported printer or that emulate several data streams. It is simpler to make the necessary changes for these printers by editing the printer colon file. See Adding a Printer Using the Printer Colon File in *Printers and printing*.

"Adding an Unsupported Device to the System" on page 104 offers an overview of the major steps required to add an unsupported device of any type to your system.

# Adding a New Printer Type to Your System

To add an unsupported printer to your system, you must add a new Printer definition to the printer directories. For more complicated scenarios, you might also need to add a new printer-specific formatter to the printer backend.

Example of Print Formatter in *Printers and printing* shows how the print formatter interacts with the printer formatter subroutines.

## Additional Steps for Adding a New Printer Type

However, if you want the new Printer definition to carry the name of the new printer, you must develop a new Predefined definition to carry the new printer information besides adding a new Printer definition.

Use the **piopredef** command to develop a new Predefined definition to carry the new printer information.

Steps for adding a new printer-specific formatter to the printer backend are discussed in Adding a Printer Formatter to the Printer Backend . Example of Print Formatter in *Printers and printing* shows how print formatters can interact with the printer formatter subroutines.

**Note:** These instructions apply to the addition of a new printer definition to the system, not to the addition of a physical printer device itself. For information on adding a new printer device, refer to device configuration and management. If your new printer requires an interface other than the parallel or serial interface provided by the operating system, you must also provide a new device driver.

If the printer being added does not emulate a supported printer or if it emulates several data streams, you need to make more changes to the Printer definition. It is simpler to make the necessary changes for these printers by editing the printer colon file. See Adding a Printer Using the Printer Colon File in *Printers and printing*.

## Modifying Printer Attributes

Edit the customized file (**/var/spool/lpd/pio/custom /var/spool/lpd/pio/@local/custom** `QueueName:QueueDeviceName`), adding or changing the printer attributes to match the new printer.

For example, assume that you created a new file based on the existing 4201-3 printer. The customized file for the 4201-3 printer contains the following template that the printer formatter uses to initialize the printer:

```
%I[ez,em,eA,cv,eC,eO,cp,cc, . . .
```

The formatter fills in the string as directed by this template and sends the resulting sequence of commands to the 4201-3 printer. Specifically, this generates a string of escape sequences that initialize the printer and set such parameters as vertical and horizontal spacing and page length. You would construct a similar command string to properly initialize the new printer and put it into 4201-emulation mode. Although many of the escape sequences might be the same, at least one will be different: the escape sequence that is the command to put the printer into the specific printer-emulation mode. Assume that you added an **ep** attribute that specifies the string to initialize the printer to 4201-3 emulation mode, as follows:

```
\033\012\013
```

The Printer Initialization field will then be:

```
%I[ep,ez,em,eA,cv,eC,eO,cp,cc, . . .
```

You must create a virtual printer for each printer-emulation mode you want to use. See Real and Virtual Printers in *Printers and printing*.

## Adding a Printer Definition

To add a new printer to the system, you must first create a description of the printer by adding a new printer definition to the printer definition directories.

Typically, to add a new printer definition to the database, you first modify an existing printer definition and then create a customized printer definition in the Customized Printer Directory.

Once you have added the new customized printer definition to the directory, the **mkvirprt** command uses it to present the new printer as a choice for printer addition and selection. Because the new printer definition is a customized printer definition, it appears in the list of printers under the name of the original printer from which it was customized.

A totally new printer must be added as a predefined printer definition in the **/usr/lib/lpd/pio/predef** directory. If the user chooses to work with `printers` once this new predefined printer definition is added to the Predefined Printer Directory, the **mkvirprt** command can then list all the printers in that directory. The added printer appears on the list of printers given to the user as if it had been supported all along. Specific information about this printer can then be extended, added, modified, or deleted, as necessary.

Printer Support in *Printers and printing* lists the supported printer types and names of representative printers.

## Adding a Printer Formatter to the Printer Backend

If your new printer's data stream differs significantly from one of the numerous printer data streams currently handled by the operating system, you must define a new backend formatter.

Adding a new formatter does not require the addition of a new backend. Instead, all you typically need are modifications to the formatter commands associated with that printer under the supervision of the existing printer backend. If a new backend is required, see Printer Backend Overview for Programming in *Printers and printing*.

## Understanding Embedded References in Printer Attribute Strings

The attribute string retrieved by the **piocmdout**, **piogetstr**, and **piogetvals** subroutines can contain embedded references to other attribute strings or integers.

The attribute string can also contain embedded logic that dynamically determines the content of the constructed string. This allows the constructed string to reflect the state of the formatter environment when one of these subroutines is called.

Embedded references and logic are defined with escape sequences that are placed at appropriate locations in the attribute string. The first character of each escape sequence is always the **%** character. This character indicates the beginning of an escape sequence. The second character (and sometimes subsequent characters) define the operation to be performed. The remainder of the characters (if any) in the escape sequence are operands to be used in performing the specified operation.

The escape sequences that can be specified in an attribute string are based on the **terminfo** parameterized string escape sequences for terminals. These escape sequences have been modified and extended for printers.

The attribute names that can be referenced by attribute strings are:

- The names of all attribute variables (which can be integer or string variables) defined to the **piogetvals** subroutine. When references are made to these variables, the **piogetvals**-defined versions are the values used.
- All other attributes names in the database. These attributes are considered string constants.

Any attribute value (integer variable, string variable, or string constant) can be referenced by any attribute string. Consequently, it is important that the formatter ensures that the values for all the integer variables and string variables defined to the **piogetvals** subroutine are kept current.

The formatter must not assume that the particular attribute string whose name it specifies to the **piogetstr** or **piocmdout** subroutine does not reference certain variables. The attribute string is retrieved

from the database that is external to the formatter. The values in the database represented by the string can be changed to reference additional variables without the formatter's knowledge.

# Small Computer System Interface Subsystem (Parallel SCSI)

This overview describes the interface between a small computer system interface (SCSI) device driver and a SCSI adapter device driver. The information in the article is specific for the parallel SCSI implementation.

Parallel SCSI is the traditional physical transport that was originally described in the SCSI-1 and SCSI-2 standards, culminating in support of Ultra 320. In the original implementation, SCSI was both a physical transport (parallel bus) and a logical command and response protocol. In the following topics, the term SCSI is used only for the traditional, parallel bus implementation of SCSI. The following topics can help you design and write a parallel SCSI device driver that can interface with an existing parallel SCSI adapter device driver. You can also use the following topics to design and write a parallel SCSI adapter device driver that interfaces with existing parallel SCSI device drivers.

For information about the implementation of SCSI on Fibre Channel (FCP), iSCSI and SAS transport types, see SCSI Architectural Model Subsystem.

**Related concepts**

Logical File System Kernel Services
The Logical File System services (also known as the **fp_**services) allow processes running in kernel mode to open and manipulate files in the same way that user-mode processes do. Data access limitations make it unreasonable to accomplish these tasks with system calls, so a subset of the file system calls has been provided with an alternate kernel-only interface.

**Related information**

scdisk SCSI Device Driver
scsidisk SCSI Device Driver
SCSI Adapter Device Driver
SCIOCMD SCSI Adapter Device Driver ioctl Operation
SCIODIAG (Diagnostic) SCSI Adapter Device Driver ioctl Operation
SCIODNLD (Download) SCSI Adapter Device Driver ioctl Operation
SCIOEVENT (Event) SCSI Adapter Device Driver ioctl Operation
SCIOGTHW (Gathered Write) SCSI Adapter Device Driver ioctl Operation
SCIOHALT (HALT) SCSI Adapter Device Driver ioctl Operation
SCIOINQU (Inquiry) SCSI Adapter Device Driver ioctl Operation
SCIOREAD (Read) SCSI Adapter Device Driver ioctl Operation
SCIORESET (Reset) SCSI Adapter Device Driver ioctl Operation
SCIOSTART (Start SCSI) SCSI Adapter Device Driver ioctl Operation
SCIOSTARTTGT (Start Target) SCSI Adapter Device Driver ioctl Operation
SCIOSTOP (Stop Device) SCSI Adapter Device Driver ioctl Operation
SCIOSTOPTGT (Stop Target) SCSI Adapter Device Driver ioctl Operation
SCIOSTUNIT (Start Unit) SCSI Adapter Device Driver ioctl Operation
SCIOTRAM (Diagnostic) SCSI Adapter Device Driver ioctl Operation
SCIOTUR (Test Unit Ready) SCSI Adapter Device Driver ioctl Operation

## SCSI Subsystem Overview

This section frequently refers to both a *SCSI device driver* and a *SCSI adapter device driver*. These two distinct device drivers work together in a layered approach to support attachment of a range of SCSI devices.

The SCSI adapter device driver is the *lower* device driver of the pair, and the SCSI device driver is the *upper* device driver.

## Responsibilities of the SCSI Adapter Device Driver

The SCSI adapter device driver (the lower layer) is the software interface to the system hardware. This hardware includes the SCSI bus hardware plus any other system I/O hardware required to run an I/O request. The SCSI adapter device driver hides the details of the I/O hardware from the SCSI device driver. The design of the software interface allows a user with limited knowledge of the system hardware to write the upper device driver.

The SCSI adapter device driver manages the SCSI bus but not the SCSI devices. It can send and receive SCSI commands, but it cannot interpret the contents of the commands. The lower driver also provides recovery and logging for errors related to the SCSI bus and system I/O hardware. Management of the device specifics is left to the SCSI device driver. The interface of the two drivers allows the upper driver to communicate with different SCSI bus adapters without requiring special code paths for each adapter.

## Responsibilities of the SCSI Device Driver

The SCSI device driver (the upper layer) provides the rest of the operating system with the software interface to a given SCSI device or device class. The upper layer recognizes which SCSI commands are required to control a particular SCSI device or device class.

The SCSI device driver builds I/O requests containing device SCSI commands and sends them to the SCSI adapter device driver in the sequence needed to operate the device successfully. The SCSI device driver cannot manage adapter resources or give the SCSI command to the adapter. Specifics about the adapter and system hardware are left to the lower layer.

The SCSI device driver also provides recovery and logging for errors related to the SCSI device it controls.

The operating system provides several kernel services allowing the SCSI device driver to communicate with SCSI adapter device driver entry points without having the actual name or address of those entry points. The description contained in “Logical File System Kernel Services” on page 65 can provide more information.

## Communication between SCSI Devices

When two SCSI devices communicate, one assumes the initiator-mode role, and the other assumes the target-mode role.

The initiator-mode device generates the SCSI command, which requests an operation, and the target-mode device receives the SCSI command and acts. It is possible for a SCSI device to perform both roles simultaneously.

When writing a new SCSI adapter device driver, the writer must know which mode or modes must be supported to meet the requirements of the SCSI adapter and any interfaced SCSI device drivers. When a SCSI adapter device driver is added so that a new SCSI adapter works with all existing SCSI device drivers, both initiator-mode and target-mode must be supported in the SCSI adapter device driver.

### *Initiator-Mode Support*

The interface between the SCSI device driver and the SCSI adapter device driver for initiator-mode support (that is, the attached device acts as a target) is accessed through calls to the SCSI adapter device driver **open**, **close**, **ioctl**, and **strategy** routines. I/O requests are queued to the SCSI adapter device driver through calls to its strategy entry point.

Communication between the SCSI device driver and the SCSI adapter device driver for a particular initiator I/O request is made through the **sc_buf** structure, which is passed to and from the strategy routine in the same way a standard driver uses a **struct buf** structure.

### *Target-Mode Support*

The interface between the SCSI device driver and the SCSI adapter device driver for target-mode support (that is, the attached device acts as an initiator) is accessed through calls to the SCSI adapter device driver **open**, **close**, and **ioctl** subroutines. Buffers that contain data received from an attached initiator

device are passed from the SCSI adapter device driver to the SCSI device driver, and back again, in **tm_buf** structures.

Communication between the SCSI adapter device driver and the SCSI device driver for a particular data transfer is made by passing the **tm_buf** structures by pointer directly to routines whose entry points have been previously registered. This registration occurs as part of the sequence of commands the SCSI device driver executes using calls to the SCSI adapter device driver when the device driver opens a target-mode device instance.

## Understanding SCSI Asynchronous Event Handling

A SCSI device driver can register a particular device instance for receiving asynchronous event status by calling the **SCIOEVENT** ioctl operation for the SCSI-adapter device driver. When an event covered by the **SCIOEVENT** ioctl operation is detected by the SCSI adapter device driver, it builds an **sc_event_info** structure and passes a pointer to the structure and to the asynchronous event-handler routine entry point, which was previously registered.

**Note:** This operation is not supported by all SCSI I/O controllers.

The fields in the structure are filled in by the SCSI adapter device driver as follows:

| Item | Description |
|---|---|
| **id** | For initiator mode, this is set to the SCSI ID of the attached SCSI target device. For target mode, this is set to the SCSI ID of the attached SCSI initiator device. |
| **lun** | For initiator mode, this is set to the SCSI LUN of the attached SCSI target device. For target mode, this is set to 0). |
| **mode** | Identifies whether the initiator or target mode device is being reported. The following values are possible:<br>**SC_IM_MODE**<br>An initiator mode device is being reported.<br>**SC_TM_MODE**<br>A target mode device is being reported. |
| **events** | This field is set to indicate what event or events are being reported. The following values are possible, as defined in the **/usr/include/sys/scsi.h** file:<br>**SC_FATAL_HDW_ERR**<br>A fatal adapter hardware error occurred.<br>**SC_ADAP_CMD_FAILED**<br>An unrecoverable adapter command failure occurred.<br>**SC_SCSI_RESET_EVENT**<br>A SCSI bus reset was detected.<br>**SC_BUFS_EXHAUSTED**<br>In target-mode, a maximum buffer usage event has occurred. |
| **adap_devno** | This field is set to indicate the device major and minor numbers of the adapter on which the device is located. |
| **async_correlator** | This field is set to the value passed to the SCSI adapter device driver in the **sc_event_struct** structure. The SCSI device driver may optionally use this field to provide an efficient means of associating event status with the device instance it goes with. Alternatively, the SCSI device driver uses the combination of the `id`, `lun`, `mode`, and `adap_devno` fields to identify the device instance. |

**Note:** Reserved fields should be set to 0 by the SCSI adapter device driver.

The information reported in the `sc_event_info.events` field does not queue to the SCSI device driver, but is instead reported as one or more flags as they occur. Because the data does not queue, the SCSI adapter device driver writer can use a single **sc_event_info** structure and pass it one at a time, by pointer, to each asynchronous event handler routine for the appropriate device instance. After determining for which device the events are being reported, the SCSI device driver must copy the `sc_event_info.events` field into local space and must not modify the contents of the rest of the **sc_event_info** structure.

Because the event status is optional, the SCSI device driver writer determines what action is necessary to take upon receiving event status. The writer may decide to save the status and report it back to the calling application, or the SCSI device driver or application level program can take error recovery actions.

## Defined Events and Recovery Actions

The adapter fatal hardware failure event is intended to indicate that no further commands to or from this SCSI device are likely to succeed, because the adapter it is attached to has failed. It is recommended that the application end the session with the device.

The unrecoverable adapter command failure event is not necessarily a fatal condition, but it can indicate that the adapter is not functioning properly. Possible actions by the application program include:

- Ending of the session with the device in the near future
- Ending of the session after multiple (two or more) such events
- Attempting to continue the session indefinitely

The SCSI Bus Reset detection event is mainly intended as information only, but may be used by the application to perform further actions, if necessary.

The maximum buffer usage detected event applies only to a given target-mode device; it will not be reported for an initiator-mode device. This event indicates to the application that this particular target-mode device instance has filled its maximum allotted buffer space. The application should perform **read** system calls fast enough to prevent this condition. If this event occurs, data is not lost, but it is delayed to prevent further buffer usage. Data reception will be restored when the application empties enough buffers to continue reasonable operations. The **num_bufs** attribute may need to be increased to help minimize this problem. Also, it is possible that regardless of the number of buffers, the application simply is not processing received data fast enough. This may require some fine tuning of the application's data processing routines.

## Asynchronous Event-Handling Routine

The SCSI-device driver asynchronous event-handling routine is typically called directly from the hardware interrupt-handling routine for the SCSI adapter device driver. The SCSI device driver writer must be aware of how this affects the design of the SCSI device driver.

Because the event handling routine is running on the hardware interrupt level, the SCSI device driver must be careful to limit operations in that routine. Processing should be kept to a minimum. In particular, if any error recovery actions are performed, it is recommended that the event-handling routine set state or status flags only and allow a process level routine to perform the actual operations.

The SCSI device driver must be careful to disable interrupts at the correct level in places where the SCSI device driver's lower execution priority routines manipulate variables that are also modified by the event-handling routine. To allow the SCSI device driver to disable at the correct level, the SCSI adapter device driver writer must provide a configuration database attribute that defines the interrupt class, or priority, it runs on. This attribute must be named **intr_priority** so that the SCSI device driver configuration method knows which attribute of the parent adapter to query. The SCSI device driver configuration method should then pass this interrupt priority value to the SCSI device driver along with other configuration data for the device instance.

The SCSI device driver writer must follow any other general system rules for writing a routine that must execute in an interrupt environment. For example, the routine must not attempt to sleep or wait on I/O operations. It can perform wakeups to allow the process level to handle those operations.

Because the SCSI device driver copies the information from the **sc_event_info.events** field on each call to its asynchronous event-handling routine, there is no resource to free or any information which must be passed back later to the SCSI adapter device driver.

## SCSI Error Recovery

The SCSI error-recovery process handles different issues depending on whether the SCSI device is in initiator mode or target mode.

If the device is in initiator mode, the error-recovery process varies depending on whether or not the device is supporting command queuing.

### SCSI Initiator-Mode Recovery When Not Command Tag Queuing

If an error such as a check condition or hardware failure occurs, transactions queued within the SCSI adapter device driver are terminated abnormally with **iodone** calls. The transaction active during the error is returned with the `sc_buf.bufstruct.b_error` field set to **EIO**.

Other transactions in the queue are returned with the `sc_buf.bufstruct.b_error` field set to **ENXIO**. The SCSI device driver should process or recover the condition, rerunning any mode selects or device reservations to recover from this condition properly. After this recovery, it should reschedule the transaction that had the error. In many cases, the SCSI device driver only needs to retry the unsuccessful operation.

The SCSI adapter device driver should never retry a SCSI command on error after the command has successfully been given to the adapter. The consequences for retrying a SCSI command at this point range from minimal to catastrophic, depending upon the type of device. Commands for certain devices cannot be retried immediately after a failure (for example, tapes and other sequential access devices). If such an error occurs, the failed command returns an appropriate error status with an **iodone** call to the SCSI device driver for error recovery. Only the SCSI device driver that originally issued the command knows if the command can be retried on the device. The SCSI adapter device driver must only retry commands that were never successfully transferred to the adapter. In this case, if retries are successful, the **sc_buf** status should not reflect an error. However, the SCSI adapter device driver should perform error logging on the retried condition.

The first transaction passed to the SCSI adapter device driver during error recovery must include a special flag. This **SC_RESUME** flag in the `sc_buf.flags` field must be set to inform the SCSI adapter device driver that the SCSI device driver has recognized the fatal error and is beginning recovery operations. Any transactions passed to the SCSI adapter device driver, after the fatal error occurs and before the **SC_RESUME** transaction is issued, should be flushed; that is, returned with an error type of **ENXIO** through an **iodone** call.

**Note:** If a SCSI device driver continues to pass transactions to the SCSI adapter device driver after the SCSI adapter device driver has flushed the queue, these transactions are also flushed with an error return of **ENXIO** through the **iodone** service. This gives the SCSI device driver a positive indication of all transactions flushed.

If the SCSI device driver is executing a gathered write operation, the error-recovery information mentioned previously is still valid, but the caller must restore the contents of the `sc_buf.resvdw1` field and the **uio** struct that the field pointed to before attempting the retry. The retry must occur from the SCSI device driver's process level; it cannot be performed from the caller's **iodone** subroutine. Also, additional return codes of **EFAULT** and **ENOMEM** are possible in the `sc_buf.bufstruct.b_error` field for a gathered write operation.

### SCSI Initiator-Mode Recovery During Command Tag Queuing

If the SCSI device driver is queuing multiple transactions to the device and either a check condition error or a command terminated error occurs, the SCSI adapter driver does not clear all transactions in its queues for the device.

It returns the failed transaction to the SCSI device driver with an indication that the queue for this device is not cleared by setting the **SC_DID_NOT_CLEAR_Q** flag in the `sc_buf.adap_q_status` field. The SCSI

adapter driver halts the queue for this device awaiting error recovery notification from the SCSI device driver. The SCSI device driver then has three options to recover from this error:

- Send one error recovery command (request sense) to the device.
- Clear the SCSI adapter driver's queue for this device.
- Resume the SCSI adapter driver's queue for this device.

When the SCSI adapter driver's queue is halted, the SCSI device drive can get sense data from a device by setting the **SC_RESUME** flag in the `sc_buf.flags` field and the **SC_NO_Q** flag in `sc_buf.q_tag_msg` field of the request-sense **sc_buf**. This action notifies the SCSI adapter driver that this is an error-recovery transaction and should be sent to the device while the remainder of the queue for the device remains halted. When the request sense completes, the SCSI device driver needs to either clear or resume the SCSI adapter driver's queue for this device.

The SCSI device driver can notify the SCSI adapter driver to clear its halted queue by sending a transaction with the **SC_Q_CLR** flag in the `sc_buf.flags` field. This transaction must not contain a SCSI command because it is cleared from the SCSI adapter driver's queue without being sent to the adapter. However, this transaction must have the SCSI ID field (`sc_buf.scsi_command.scsi_id`) and the LUN fields (`sc_buf.scsi_command.scsi_cmd.lun` and `sc_buf.lun`) filled in with the device's SCSI ID and logical unit number (LUN). If addressing LUNs 8 - 31, the `sc_buf.lun` field should be set to the logical unit number and the `sc_buf.scsi_command.scsi_cmd.lun` field should be zeroed out. See the descriptions of these fields for further explanation. Upon receiving an **SC_Q_CLR** transaction, the SCSI adapter driver flushes all transactions for this device and sets their `sc_buf.bufstruct.b_error` fields to **ENXIO**. The SCSI device driver must wait until the **sc_buf** with the **SC_Q_CLR** flag set is returned before it resumes issuing transactions. The first transaction sent by the SCSI device driver after it receives the returned **SC_Q_CLR** transaction must have the **SC_RESUME** flag set in the sc_buf.flags fields.

If the SCSI device driver wants the SCSI adapter driver to resume its halted queue, it must send a transaction with the **SC_Q_RESUME** flag set in the sc_buf.flags field. This transaction can contain an actual SCSI command, but it is not required. However, this transaction must have the `sc_buf.scsi_command.scsi_id`, `sc_buf.scsi_command.scsi_cmd.lun`,and the `sc_buf.lun` fields filled in with the device's SCSI ID and logical unit number. See the description of these fields for further details. If this is the first transaction issued by the SCSI device driver after receiving the error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set as well as the **SC_Q_RESUME** flag.

## Analyzing Returned Status

The order of precedence should be followed by SCSI device drivers when analyzing the returned status.

1. If the `sc_buf.bufstruct.b_flags` field has the **B_ERROR** flag set, then an error has occurred and the `sc_buf.bufstruct.b_error` field contains a valid **errno** value.

   If the `b_error` field contains the **ENXIO** value, either the command needs to be restarted or it was canceled at the request of the SCSI device driver.

   If the `b_error` field contains the **EIO** value, then either one or no flag is set in the `sc_buf.status_validity` field. If a flag is set, an error in either the `scsi_status` or `general_card_status` field is the cause.

   If the `status_validity` field is 0, then the `sc_buf.bufstruct.b_resid` field should be examined to see if the SCSI command issued was in error. The `b_resid` field can have a value without an error having occurred. To decide whether an error has occurred, the SCSI device driver must evaluate this field with regard to the SCSI command being sent and the SCSI device being driven.

   If the SCSI device driver is queuing multiple transactions to the device and if either **SC_CHECK_CONDITION** or **SC_COMMAND_TERMINATED** is set in `scsi_status` , then the value of `sc_buf.adap_q_status` must be analyzed to determine if the adapter driver has cleared its queue for this device. If the SCSI adapter driver has not cleared its queue after an error, then it holds that queue in a halted state.

If `sc_buf.adap_q_status` is set to 0, the SCSI adapter driver has cleared its queue for this device and any transactions outstanding are flushed back to the SCSI device driver with an error of **ENXIO**.

If the **SC_DID_NOT_CLEAR_Q** flag is set in the `sc_buf.adap_q_status` field, the adapter driver has not cleared its queue for this device. When this condition occurs, the SCSI adapter driver allows the SCSI device driver to send one error recovery transaction (request sense) that has the field `sc_buf.q_tag_msg` set to **SC_NO_Q** and the field `sc_buf.flags` set to **SC_RESUME**. The SCSI device driver can then notify the SCSI adapter driver to clear or resume its queue for the device by sending a **SC_Q CLR** or **SC_Q_RESUME** transaction.

If the SCSI device driver does not queue multiple transactions to the device (that is, the **SC_NO_Q** is set in `sc_buf.q_tag_msg` ), then the SCSI adapter clears its queue on error and sets `sc_buf.adap_q_status` to 0.

2. If the `sc_buf.bufstruct.b_flags` field does not have the **B_ERROR** flag set, then no error is being reported. However, the SCSI device driver should examine the `b_resid` field to check for cases where less data was transferred than expected. For some SCSI commands, this occurrence might not represent an error. The SCSI device driver must determine if an error has occurred.

   If a nonzero `b_resid` field does represent an error condition, then the device queue is not halted by the SCSI adapter device driver. It is possible for one or more succeeding queued commands to be sent to the adapter (and possibly the device). Recovering from this situation is the responsibility of the SCSI device driver.
3. In any of the above cases, if `sc_buf.bufstruct.b_flags` field has the **B_ERROR** flag set, then the queue of the device in question has been halted. The first **sc_buf** structure sent to recover the error (or continue operations) must have the **SC_RESUME** bit set in the `sc_buf.flags` field.

### Target-Mode Error Recovery

If an error occurs during the reception of **send** command data, the SCSI adapter device driver sets the **TM_ERROR** flag in the `tm_buf.user_flag` field. The SCSI adapter device driver also sets the **SC_ADAPTER_ERROR** bit in the `tm_buf.status_validity` field and sets a single flag in the `tm_buf.general_card_status` field to indicate the error that occurred.

In the SCSI subsystem, an error during a **send** command does not affect future target-mode data reception. Future **send** commands continue to be processed by the SCSI adapter device driver and queue up, as necessary, after the data with the error. The SCSI device driver continues processing the **send** command data, satisfying user read requests as usual except that the error status is returned for the appropriate user request. Any error recovery or synchronization procedures the user requires for a target-mode received-data error must be implemented in user-supplied software.

## A Typical Initiator-Mode SCSI Driver Transaction Sequence

A simplified sequence of events for a transaction between a SCSI device driver and a SCSI adapter device driver follows. In this sequence, routine names preceded by a **dd_** are part of the SCSI device driver, where as those preceded by a **sc_** are part of the SCSI adapter device driver.

1. The SCSI device driver receives a call to its **dd_strategy** routine; any required internal queuing occurs in this routine. The **dd_strategy** entry point then triggers the operation by calling the **dd_start** entry point. The **dd_start** routine invokes the **sc_strategy** entry point by calling the **devstrategy** kernel service with the relevant **sc_buf** structure as a parameter.
2. The **sc_strategy** entry point initially checks the **sc_buf** structure for validity. These checks include validating the `devno` field, matching the SCSI ID/LUN to internal tables for configuration purposes, and validating the request size.
3. Although the SCSI adapter device driver cannot reorder transactions, it does perform queue chaining. If no other transactions are pending for the requested device, the **sc_strategy** routine immediately calls the **sc_start** routine with the new transaction. If there are other transactions pending, the new transaction is added to the tail of the device chain.

4. At each interrupt, the **sc_intr** interrupt handler verifies the current status. The SCSI adapter device driver fills in the `sc_buf status_validity` field, updating the `scsi_status` and `general_card_status` fields as required.
5. The SCSI adapter device driver also enters the `bufstruct.b_resid` field with the number of bytes not transferred from the request. If all the data was transferred, the `b_resid` field is set to a value of 0. When a transaction completes, the **sc_intr** routine causes the **sc_buf** entry to be removed from the device queue and calls the **iodone** kernel service, passing the just dequeued **sc_buf** structure for the device as the parameter.

   The **sc_start** routine is then called again to process the next transaction on the device queue. The **iodone** kernel service calls the SCSI device driver **dd_iodone** entry point, signaling the SCSI device driver that the particular transaction has completed.
6. The SCSI device driver **dd_iodone** routine investigates the I/O completion codes in the **sc_buf** status entries and performs error recovery, if required. If the operation completed correctly, the SCSI device driver dequeues the original buffer structures. It calls the **iodone** kernel service with the original buffer pointers to notify the originator of the request.

## Understanding SCSI Device Driver Internal Commands

During initialization, error recovery, and open or close operations, SCSI device drivers initiate some transactions not directly related to an operating system request. These transactions are called *internal commands* and are relatively simple to handle.

Internal commands differ from operating system-initiated transactions in several ways. The primary difference is that the SCSI device driver is required to generate a **struct buf** that is not related to a specific request. Also, the actual SCSI commands are typically more control-oriented than data transfer-related.

The only special requirement for commands with short data-phase transfers (less than or equal to 256 bytes) is that the SCSI device driver must have pinned the memory being transferred into or out of system memory pages. However, due to system hardware considerations, additional precautions must be taken for data transfers into system memory pages when the transfers are larger than 256 bytes. The problem is that any system memory area with a DMA data operation in progress causes the entire memory page that contains it to become inaccessible.

As a result, a SCSI device driver that initiates an internal command with more than 256 bytes must have preallocated and pinned an area of some multiple whose size is the system page size. The driver must not place in this area any other data areas that it may need to access while I/O is being performed into or out of that page. Memory pages so allocated must be avoided by the device driver from the moment the transaction is passed to the adapter device driver until the device driver **iodone** routine is called for the transaction (and for any other transactions to those pages).

## Understanding the Execution of Initiator I/O Requests

During normal processing, many transactions are queued in the SCSI device driver.

As the SCSI device driver processes these transactions and passes them to the SCSI adapter device driver, the SCSI device driver moves them to the in-process queue. When the SCSI adapter device driver returns through the **iodone** service with one of these transactions, the SCSI device driver either recovers any errors on the transaction or returns using the **iodone** kernel service to the calling level.

The SCSI device driver can send only one **sc_buf** structure per call to the SCSI adapter device driver. Thus, the **sc_buf.bufstruct.av_forw** pointer should be null when given to the SCSI adapter device driver, which indicates that this is the only request. The SCSI device driver can queue multiple **sc_buf** requests by making multiple calls to the SCSI adapter device driver strategy routine.

## Spanned (Consolidated) Commands

Some kernel operations might be composed of sequential operations to a device. For example, if consecutive blocks are written to disk, blocks might or might not be in physically consecutive buffer pool blocks.

To enhance SCSI bus performance, the SCSI device driver should consolidate multiple queued requests when possible into a single SCSI command. To allow the SCSI adapter device driver the ability to handle the scatter and gather operations required, the **sc_buf.bp** should always point to the first **buf** structure entry for the spanned transaction. A null-terminated list of additional **struct buf** entries should be chained from the first field through the `buf.av_forw` field to give the SCSI adapter device driver enough information to perform the DMA scatter and gather operations required. This information must include at least the buffer's starting address, length, and cross-memory descriptor.

The spanned requests should always be for requests in either the read or write direction but not both, because the SCSI adapter device driver must be given a single SCSI command to handle the requests. The spanned request should always consist of complete I/O requests (including the additional **struct buf** entries). The SCSI device driver should not attempt to use partial requests to reach the maximum transfer size.

The maximum transfer size is actually adapter-dependent. The **IOCINFO** ioctl operation can be used to discover the SCSI adapter device driver's maximum allowable transfer size. To ease the design, implementation, and testing of components that might need to interact with multiple SCSI-adapter device drivers, a required minimum size has been established that all SCSI adapter device drivers must be capable of supporting. The value of this minimum/maximum transfer size is defined as the following value in the **/usr/include/sys/scsi.h** file:

```
SC_MAXREQUEST           /* maximum transfer request for a single */
                        /* SCSI command (in bytes) */
```

If a transfer size larger than the supported maximum is attempted, the SCSI adapter device driver returns a value of **EINVAL** in the `sc_buf.bufstruct.b_error` field.

Due to system hardware requirements, the SCSI device driver must consolidate only commands that are memory page-aligned at both their starting and ending addresses. Specifically, this applies to the consolidation of *inner* memory buffers. The ending address of the first buffer and the starting address of all subsequent buffers should be memory page-aligned. However, the starting address of the first memory buffer and the ending address of the last do not need to be aligned so.

The purpose of consolidating transactions is to decrease the number of SCSI commands and bus phases required to perform the required operation. The time required to maintain the simple chain of **buf** structure entries is significantly less than the overhead of multiple (even two) SCSI bus transactions.

## Fragmented Commands

Single I/O requests larger than the maximum transfer size must be divided into smaller requests by the SCSI device driver.

For calls to a SCSI device driver's character I/O (read/write) entry points, the **uphysio** kernel service can be used to break up these requests. For a *fragmented command* such as this, the `sc_buf.bp` field should be null so that the SCSI adapter device driver uses only the information in the **sc_buf** structure to prepare for the DMA operation.

## Gathered Write Commands

The gathered write commands facilitate communication applications that are required to send header and trailer messages with data buffers. These headers and trailers are typically the same or similar for each transfer. Therefore, there might be a single copy of these messages but multiple data buffers.

The gathered write commands, accessed through the `sc_buf.resvd1` field, differ from the spanned commands, accessed through the `sc_buf.bp` field, in several ways:

- Gathered write commands can transfer data regardless of address alignment, where as spanned commands must be memory page-aligned in address and length, making small transfers difficult.
- Gathered write commands can be implemented either in software (which requires the extra step of copying the data to temporary buffers) or hardware. Spanned commands can be implemented in system hardware due to address-alignment requirements. As a result, spanned commands are potentially faster to run.
- Gathered write commands are not able to handle read requests. Spanned commands can handle both read and write requests.
- Gathered write commands can be initiated only on the process level, but spanned commands can be initiated on either the process or interrupt level.

To execute a gathered write command, the SCSI device driver must:

- Fill in the `resvd1` field with a pointer to the **uio** struct
- Call the SCSI adapter device driver on the same process level with the **sc_buf** structure in question
- Be attempting a write
- Not have put a non-null value in the `sc_buf.bp` field

If any of these conditions are not met, the gathered write commands do not succeed and the `sc_buf.bufstruct.b_error` is set to **EINVAL**.

This interface allows the SCSI adapter device driver to perform the gathered write commands in both software or and hardware as long as the adapter supports this capability. Because the gathered write commands can be performed in software (by using such kernel services as **uiomove**), the contents of the `resvd1` field and the **uio** struct can be altered. Therefore, the caller must restore the contents of both the `resvd1` field and the **uio** struct before attempting a retry. Also, the retry must occur from the process level; it must not be performed from the caller's **iodone** subroutine.

To support SCSI adapter device drivers that perform the gathered write commands in software, additional return values in the `sc_buf.bufstruct.b_error` field are possible when gathered write commands are unsuccessful.

| Item | Description |
|---|---|
| **ENOMEM** | Error due to lack of system memory to perform copy. |
| **EFAULT** | Error due to memory copy problem. |

> **Note:** The gathered write command facility is optional for both the SCSI device driver and the SCSI adapter device driver. Attempting a gathered write command to a SCSI adapter device driver that does not support gathered write can cause a system crash. Therefore, any SCSI device driver must issue a **SCIOGTHW** ioctl operation to the SCSI adapter device driver before using gathered writes. A SCSI adapter device driver that supports gathered writes must support the **SCIOGTHW** ioctl as well. The ioctl returns a successful return code if gathered writes are supported. If the ioctl fails, the SCSI device driver must not attempt a gathered write. Typically, a SCSI device driver places the **SCIOGTHW** call in its open routine for device instances that it will send gathered writes to.

## SCSI Command Tag Queuing

SCSI command tag queuing refers to queuing multiple commands to a SCSI device.

**Note:** This operation is not supported by all SCSI I/O controllers.

Queuing to the SCSI device can improve performance because the device itself determines the most efficient way to order and process commands. SCSI devices that support command tag queuing can be divided into two classes: those that clear their queues on error and those that do not. Devices that do not clear their queues on error resume processing of queued commands when the error condition is cleared

typically by receiving the next command. Devices that do clear their queues flush all commands currently outstanding.

Command tag queueing requires the SCSI adapter, the SCSI device, the SCSI device driver, and the SCSI adapter driver to support this capability. For a SCSI device driver to queue multiple commands to a SCSI device (that supports command tag queuing), it must be able to provide at least one of the following values in the `sc_buf.q_tag_msg`: **SC_SIMPLE_Q**, **SC_HEAD_OF_Q**, or **SC_ORDERED_Q**. The SCSI disk device driver and SCSI adapter driver do support this capability. This implementation provides some queuing-specific changeable attributes for disks that can queue commands. With this information, the disk device driver attempts to queue to the disk, first by queuing commands to the adapter driver. The SCSI adapter driver then queues these commands to the adapter, providing that the adapter supports command tag queuing. If the SCSI adapter does not support command tag queuing, then the SCSI adapter driver sends only one command at a time to the SCSI adapter and so multiple commands are not queued to the SCSI disk.

# Understanding the sc_buf Structure

The **sc_buf** structure is used for communication between the SCSI device driver and the SCSI adapter device driver during an initiator I/O request.

This structure is passed to and from the strategy routine in the same way a standard driver uses a **struct buf** structure.

## Fields in the sc_buf Structure

The **sc_buf** structure contains certain fields used to pass a SCSI command and associated parameters to the SCSI adapter device driver. Other fields within this structure are used to pass returned status back to the SCSI device driver. The **sc_buf** structure is defined in the **/usr/include/sys/scsi.h** file.

Fields in the **sc_buf** structure are used as follows:

1. Reserved fields should be set to a value of 0, except where noted.
2. The `bufstruct` field contains a copy of the standard **buf** buffer structure that documents the I/O request. Included in this structure, for example, are the buffer address, byte count, and transfer direction. The `b_work` field in the **buf** structure is reserved for use by the SCSI adapter device driver. The current definition of the **buf** structure is in the **/usr/include/sys/buf.h** include file.
3. The bp field points to the original buffer structure received by the SCSI Device Driver from the caller, if any. This can be a chain of entries in the case of spanned transfers (SCSI commands that transfer data from or to more than one system-memory buffer). A null pointer indicates a nonspanned transfer. The null value specifically tells the SCSI adapter device driver that all the information needed to perform the DMA data transfer is contained in the `bufstruct` fields of the **sc_buf** structure. If the bp field is set to a non-null value, the `sc_buf.resvd1` field must have a value of null, or else the operation is not allowed.
4. The `scsi_command` field, defined as a **scsi** structure, contains, for example, the SCSI ID, SCSI command length, SCSI command, and a flag variable:
   a. The `scsi_length` field is the number of bytes in the actual SCSI command. This is normally 6, 10, or 12 (decimal).
   b. The `scsi_id` field is the SCSI physical unit ID.
   c. The `scsi_flags` field contains the following bit flags:

      **SC_NODISC**
      Do not allow the target to disconnect during this command.

      **SC_ASYNC**
      Do not allow the adapter to negotiate for synchronous transfer to the SCSI device.

      During normal use, the SC_NODISC bit should not be set. Setting this bit allows a device executing commands to monopolize the SCSI bus. Sometimes it is desirable for a particular device to maintain control of the bus once it has successfully arbitrated for it; for instance, when this is the only device on the SCSI bus or the only device that will be in use. For performance reasons, it

might not be desirable to go through SCSI selections again to save SCSI bus overhead on each command.

Also during normal use, the SC_ASYNC bit must not be set. It should be set only in cases where a previous command to the device ended in an unexpected SCSI bus free condition. This condition is noted as **SC_SCSI_BUS_FAULT** in the `general_card_status` field of the **sc_cmd** structure. Because other errors might also result in the **SC_SCSI_BUS_FAULT** flag being set, the SC_ASYNC bit should only be set on the last retry of the failed command.

d. The **sc_cmd** structure contains the physical SCSI command block. The 6 to 12 bytes of a single SCSI command are stored in consecutive bytes, with the op code and logical unit identified individually. The **sc_cmd** structure contains the following fields:

- The `scsi_op_code` field specifies the standard SCSI op code for this command.
- The `lun` field specifies the standard SCSI logical unit for the physical SCSI device controller. Typically, there will be one LUN per controller (LUN=0, for example) for devices with imbedded controllers. Only the upper 3 bits of this field contain the actual LUN ID. If addressing LUN's 0 - 7, this lun field should always be filled in with the LUN value. When addressing LUN's 8 - 31, this lun field should be set to 0 and the LUN value should be placed into the sc_buf.lun field described in this section.
- The `scsi_bytes` field contains the remaining command-unique bytes of the SCSI command block. The actual number of bytes depends on the value in the **scsi_op_code** field.
- The `resvd1` field is set to a non-null value to indicate a request for a gathered write. A gathered write means the SCSI command conducts a system-to-device data transfer where multiple, noncontiguous system buffers contain the write data. This data is transferred in order as a single data transfer for the SCSI command in this **sc_buf** structure.

  The contents of the `resvd1` field, if non-null, must be a pointer to the **uio** structure that is passed to the SCSI device driver. The SCSI adapter device driver treats the `resvd1` field as a pointer to a **uio** structure that accesses the **iovec** structures containing pointers to the data. There are no address-alignment restrictions on the data in the **iovec** structures. The only restriction is that the total transfer length of all the data must not exceed the maximum transfer length for the adapter device driver.

  The `sc_buf.bufstruct.b_un.b_addr` field, which normally contains the starting system-buffer address, is ignored and can be altered by the SCSI adapter device driver when the **sc_buf** is returned. The `sc_buf.bufstruct.b_bcount` field should be set by the caller to the total transfer length for the data.

5. The `timeout_value` field specifies the time-out limit (in seconds) to be used for completion of this command. A time-out value of 0 means no time-out is applied to this I/O request.
6. The `status_validity` field contains an output parameter that can have one of the following bit flags as a value:

   **SC_SCSI_ERROR**
   The `scsi_status` field is valid.

   **SC_ADAPTER_ERROR**
   The `general_card_status` field is valid.
7. The `scsi_status` field in the **sc_buf** structure is an output parameter that provides valid SCSI command completion status when its **status_validity** bit is nonzero. The `sc_buf.bufstruct.b_error` field should be set to **EIO** anytime the `scsi_status` field is valid. Typical status values include:

   **SC_GOOD_STATUS**
   The target successfully completed the command.

   **SC_CHECK_CONDITION**
   The target is reporting an error, exception, or other conditions.

   **SC_BUSY_STATUS**
   The target is currently busy and cannot accept a command now.

**SC_RESERVATION_CONFLICT**
The target is reserved by another initiator and cannot be accessed.

**SC_COMMAND_TERMINATED**
The target terminated this command after receiving a terminate I/O process message from the SCSI adapter.

**SC_QUEUE_FULL**
The target's command queue is full, so this command is returned.

8. The `general_card_status` field is an output parameter that is valid when its **status_validity** bit is nonzero. The `sc_buf.bufstruct.b_error` field should be set to **EIO** anytime the `general_card_status` field is valid. This field contains generic SCSI adapter card status. It is intentionally general in coverage so that it can report error status from any typical SCSI adapter.

   If an error is detected during execution of a SCSI command, and the error prevented the SCSI command from actually being sent to the SCSI bus by the adapter, then the error should be processed or recovered, or both, by the SCSI adapter device driver.

   If it is recovered successfully by the SCSI adapter device driver, the error is logged, as appropriate, but is not reflected in the **general_card_status** byte. If the error cannot be recovered by the SCSI adapter device driver, the appropriate **general_card_status** bit is set and the **sc_buf** structure is returned to the SCSI device driver for further processing.

   If an error is detected after the command was actually sent to the SCSI device, then it should be processed or recovered, or both, by the SCSI device driver.

   For error logging, the SCSI adapter device driver logs SCSI bus- and adapter-related conditions, where as the SCSI device driver logs SCSI device-related errors. In the following description, a capital letter "A" after the error name indicates that the SCSI adapter device driver handles error logging. A capital letter "H" indicates that the SCSI device driver handles error logging.

   Some of the following error conditions indicate a SCSI device failure. Others are SCSI bus- or adapter-related.

   **SC_HOST_IO_BUS_ERR (A)**
   The system I/O bus generated or detected an error during a DMA or Programmed I/O (PIO) transfer.

   **SC_SCSI_BUS_FAULT (H)**
   The SCSI bus protocol or hardware was unsuccessful.

   **SC_CMD_TIMEOUT (H)**
   The command timed out before completion.

   **SC_NO_DEVICE_RESPONSE (H)**
   The target device did not respond to selection phase.

   **SC_ADAPTER_HDW_FAILURE (A)**
   The adapter indicated an onboard hardware failure.

   **SC_ADAPTER_SFW_FAILURE (A)**
   The adapter indicated microcode failure.

   **SC_FUSE_OR_TERMINAL_PWR (A)**
   The adapter indicated a blown terminator fuse or bad termination.

   **SC_SCSI_BUS_RESET (A)**
   The adapter indicated the SCSI bus has been reset.

9. When the SCSI device driver queues multiple transactions to a device, the `adap_q_status` field indicates whether or not the SCSI adapter driver has cleared its queue for this device after an error has occurred. The flag of **SC_DID_NOT CLEAR_Q** indicates that the SCSI adapter driver has not cleared its queue for this device and that it is in a halted state (so none of the pending queued transactions are sent to the device).
10. The `lun` field provides addressability of up to 32 logical units (LUNs). This field specifies the standard SCSI LUN for the physical SCSI device controller. If addressing LUN's 0 - 7, both this lun field (sc_buf.lun) and the lun field located in the scsi_command structure

(sc_buf.scsi_command.scsi_cmd.lun) should be set to the LUN value. If addressing LUN's 8 - 31, this lun field (sc_buf.lun) should be set to the LUN value and the lun field located in the scsi_command structure (sc_buf.scsi_command.scsi_cmd.lun) should be set to 0.

| **Logical Unit Numbers (LUNs)** | | |
|---|---|---|
| **lun Fields** | **LUN 0 - 7** | **LUN 8 - 31** |
| `sc_buf.lun` | *LUN Value* | *LUN Value* |
| `sc_buf.scsi_command.scsi_cmd.lun` | *LUN Value* | 0 |

**Note:** *LUN value* is the current value of LUN.

11. The `q_tag_msg` field indicates if the SCSI adapter can attempt to queue this transaction to the device. This information causes the SCSI adapter to fill in the Queue Tag Message Code of the queue tag message for a SCSI command. The following values are valid for this field:

    **SC_NO_Q**
    Specifies that the SCSI adapter does not send a queue tag message for this command, and so the device does not allow more than one SCSI command on its command queue. This value must be used for all commands sent to SCSI devices that do not support command tag queuing.

    **SC_SIMPLE_Q**
    Specifies placing this command in the device's command queue. The device determines the order that it executes commands in its queue. The SCSI-2 specification calls this value the "Simple Queue Tag Message."

    **SC_HEAD_OF_Q**
    Specifies placing this command first in the device's command queue. This command does not preempt an active command at the device, but it is executed before all other commands in the command queue. The SCSI-2 specification calls this value the "Head of Queue Tag Message."

    **SC_ORDERED_Q**
    Specifies placing this command in the device's command queue. The device processes these commands in the order that they are received. The SCSI-2 specification calls this value the "Ordered Queue Tag Message."

    **Note:** Commands with the value of **SC_NO_Q** for the `q_tag_msg` field (except for request sense commands) should not be queued to a device whose queue contains a command with another value for `q_tag_msg`. If commands with the **SC_NO_Q** value (except for request sense) are sent to the device, then the SCSI device driver must make sure that no active commands are using different values for `q_tag_msg`. Similarly, the SCSI device driver must also make sure that a command with a `q_tag_msg` value of **SC_ORDERED_Q**, **SC_HEAD_Q**, or **SC_SIMPLE_Q** is not sent to a device that has a command with the `q_tag_msg` field of **SC_NO_Q**.

12. The `flags` field contains bit flags sent from the SCSI device driver to the SCSI adapter device driver. The following flags are defined:

    **SC_RESUME**
    When set, means the SCSI adapter device driver should resume transaction queuing for this ID/LUN. Error recovery is complete after a **SCIOHALT** operation, check condition, or severe SCSI bus error. This flag is used to restart the SCSI adapter device driver following a reported error.

    **SC_DELAY_CMD**
    When set, means the SCSI adapter device driver should delay sending this command (following a SCSI reset or BDR to this device) by at least the number of seconds specified to the SCSI adapter device driver in its configuration information. For SCSI devices that do not require this function, this flag should not be set.

    **SC_Q_CLR**
    When set, means the SCSI adapter driver should clear its transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual SCSI command in the `sc_buf` because it is flushed back to the SCSI device driver with the rest of the transactions for this ID/LUN. However, this transaction

must have the SCSI ID field (`sc_buf.scsi_command.scsi_id`) and the LUN fields (`sc_buf.scsi_command.scsi_cmd.lun` and `sc_buf.lun`) filled in with the device's SCSI ID and logical unit number (LUN). This flag is valid only during error recovery of a check condition or command terminated at a command tag queuing device when the **SC_DID_NOT_CLR_Q** flag is set in the `sc_buf.adap_q_status` field.

**Note:** When addressing LUN's 8 - 31, be sure to see the description of the `sc_buf.lun` field within the `sc_buf` structure.

**SC_Q_RESUME**
When set, means that the SCSI adapter driver should resume its halted transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual SCSI command to be sent to the SCSI adapter driver. However, this transaction must have the `sc_buf.scsi_command.scsi_id` and `sc_buf.scsi_command.scsi_cmd.lun` fields filled in with the device's SCSI ID and logical unit number. If the transaction containing this flag setting is the first issued by the SCSI device driver after it receives an error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set also.

**Note:** When addressing LUN's 8 - 31, be sure to see the description of the `sc_buf.lun` field within the `sc_buf` structure.

# Other SCSI Design Considerations

There are some other SCSI design considerations for error recovery that are listed in this section.

## Responsibilities of the SCSI Device Driver

In the operating system, it is assumed that other SCSI initiators might be active on the SCSI bus. Usually, the SCSI device driver reserves the SCSI device at open time and releases it at close time (except when told to do otherwise through parameters in the SCSI device driver interface).

- Interfacing with block I/O and logical-volume device-driver code in the operating system.
- Translating I/O requests from the operating system into SCSI commands suitable for the particular SCSI device. These commands are then given to the SCSI adapter device driver for execution.
- Issuing any and all SCSI commands to the attached device. The SCSI adapter device driver sends no SCSI commands except those it is directed to send by the calling SCSI device driver.
- Managing SCSI device reservations and releases. Once the device is reserved, the SCSI device driver must be prepared to reserve the SCSI device again whenever a Unit Attention condition is reported through the SCSI request-sense data.

## SCSI Options to the openx Subroutine

SCSI device drivers in the operating system must support eight defined extended options in their open routine (that is, an **openx** subroutine). Additional extended options to the open are also allowed, but they must not conflict with predefined open options.

The defined extended options are bit flags in the *ext* open parameter. These options can be specified singly or in combination with each other. The required *ext* options are defined in the **/usr/include/sys/scsi.h** header file and can have one of the following values:

| Item | Description |
|---|---|
| **SC_FORCED_OPEN** | Do not honor device reservation-conflict status. |
| **SC_RETAIN_RESERVATION** | Do not release SCSI device on close. |
| **SC_DIAGNOSTIC** | Enter diagnostic mode for this device. |
| **SC_NO_RESERVE** | Prevents the reservation of the device during an **openx** subroutine call to that device. Allows multiple hosts to share a device. |
| **SC_SINGLE** | Places the selected device in Exclusive Access mode. |

| Item | Description |
| --- | --- |
| **SC_RESV_05** | Reserved for future expansion. |
| **SC_RESV_07** | Reserved for future expansion. |
| **SC_RESV_08** | Reserved for future expansion. |

## Using the SC_FORCED_OPEN Option

The **SC_FORCED_OPEN** option causes the SCSI device driver to call the SCSI adapter device driver's Bus Device Reset ioctl (**SCIORESET**) operation on the first open. This forces the device to release another initiator's reservation.

After the **SCIORESET** command is completed, other SCSI commands are sent as in a normal open. If any of the SCSI commands fail due to a reservation conflict, the open registers the failure as an **EBUSY** status. This is also the result if a reservation conflict occurs during a normal open. The SCSI device driver should require the caller to have appropriate authority to request the **SC_FORCED_OPEN** option because this request can force a device to drop a SCSI reservation. If the caller attempts to initiate this system call without the proper authority, the SCSI device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**.

## Using the SC_RETAIN_RESERVATION Option

The **SC_RETAIN_RESERVATION** option causes the SCSI device driver not to issue the SCSI release command during the close of the device. This guarantees a calling program control of the device (using SCSI reservation) through open and close cycles.

For shared devices (for example, disk or CD-ROM), the SCSI device driver must OR together this option for all opens to a given device. If any caller requests this option, the **close** routine does not issue the release even if other opens to the device do not set **SC_RETAIN_RESERVATION**. The SCSI device driver should require the caller to have appropriate authority to request the **SC_RETAIN_RESERVATION** option because this request can allow a program to monopolize a device (for example, if this is a nonshared device). If the caller attempts to initiate this system call without the proper authority, the SCSI device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**.

## Using the SC_DIAGNOSTIC Option

The **SC_DIAGNOSTIC** option causes the SCSI device driver to enter Diagnostic mode for the given device. This option directs the SCSI device driver to perform only minimal operations to open a logical path to the device. No SCSI commands should be sent to the device in the **open** or **close** routine when the device is in Diagnostic mode. One or more ioctl operations should be provided by the SCSI device driver to allow the caller to issue SCSI commands to the attached device for diagnostic purposes.

The **SC_DIAGNOSTIC** option gives the caller an exclusive open to the selected device. This option requires appropriate authority to run. If the caller attempts to initiate this system call without the proper authority, the SCSI device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**. The **SC_DIAGNOSTIC** option may be run only if the device is not already opened for normal operation. If this ioctl operation is attempted when the device is already opened, or if an **openx** call with the **SC_DIAGNOSTIC** option is already in progress, a return value of -1 should be passed, with the **errno** global variable set to a value of **EACCES**. Once successfully opened with the **SC_DIAGNOSTIC** flag, the SCSI device driver is placed in Diagnostic mode for the selected device.

## Using the SC_NO_RESERVE Option

The **SC_NO_RESERVE** option causes the SCSI device driver not to issue the SCSI reserve command during the opening of the device and not to issue the SCSI release command during the close of the device. This allows multiple hosts to share the device.

The SCSI device driver should require the caller to have appropriate authority to request the **SC_NO_RESERVE** option, because this request allows other hosts to modify data on the device. If a caller does this kind of request then the caller must ensure data integrity between multiple hosts. If the

caller attempts to initiate this system call without the proper authority, the SCSI device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**.

## Using the SC_SINGLE Option

The **SC_SINGLE** option causes the SCSI device driver to issue a normal open, but does not allow another caller to issue another open until the first caller has closed the device. This request gives the caller an exclusive open to the selected device. If this **openx** is attempted when the device is already open, a return value of -1 is passed, with the **errno** global variable set to a value of **EBUSY**.

Once sucessfully opened, the device is placed in Exclusive Access mode. If another caller tries to do any type of **open**, a return value of -1 is passed, with the **errno** global variable set to a value of **EACCES**.

The remaining options for the *ext* parameter are reserved for future requirements.

**Implementation note:** The following table shows how the various combinations of *ext* options should be handled in the SCSI device driver.

| **EXT OPTIONS openx *ext* option** | **Device Driver Action** |
|---|---|
| none | Open: normal. Close: normal. |
| diag | Open: no SCSI commands. Close: no SCSI commands. |
| diag + force | Open: issue SCIORESET otherwise, no SCSI commands issued. Close: no SCSI commands. |
| diag + force + no_reserve | Open: issue SCIORESET; otherwise, no SCSI commands isssued. Close: no SCSI commands. |
| diag + force + no_reserve + single | Open: issue SCIORESET; otherwise, no SCSI commands isssued. Close: no SCSI commands. |
| diag + force +retain | Open: issue SCIORESET; otherwise, no SCSI commands issued. Close: no SCSI commands. |
| diag + force +retain + no_reserve | Open: issue SCIORESET; otherwise, no SCSI commands issued. Close: no SCSI commands. |
| diag + force +retain + no_reserve + single | Open: issue SCIORESET; otherwise, no SCSI commands issued. Close: no SCSI commands. |
| diag + force +retain + single | Open: issue SCIORESET; otherwise, no SCSI commands issued. Close: no SCSI commands. |
| diag + force + single | Open: issue SCIORESET; otherwise, no SCSI commands issued. Close: no SCSI commands. |
| diag+no_reserve | Open: no SCSI commands. Close: no SCSI commands. |
| diag + retain | Open: no SCSI commands. Close: no SCSI commands. |
| diag + retain + no_reserve | Open: no SCSI commands. Close: no SCSI commands. |
| diag + retain + no_reserve + single | Open: no SCSI commands. Close: no SCSI commands. |
| diag + retain + single | Open: no SCSI commands. Close: no SCSI commands. |
| diag + single | Open: no SCSI commands. Close: no SCSI commands. |

| EXT OPTIONS openx *ext* option | Device Driver Action |
|---|---|
| diag + single + no_reserve | Open: no SCSI commands. Close: no SCSI commands. |
| force | Open: normal, except SCIORESET issued prior toany SCSI commands. Close: normal. |
| force + no_reserve | Open: normal except SCIORESET issued prior to any SCSI commands. No RESERVE command issued. Close: normal except no RELEASE. |
| force + retain | Open: normal, except SCIORESET issued prior to any SCSI commands. Close: no RELEASE. |
| force + retain + no_reserve | Open: normal except SCIORESET issued prior to any SCSI commands. No RESERVE command issued. Close: no RELEASE. |
| force + retain + no_reserve + single | Open: normal except SCIORESET issued prior to any SCSI commands. No RESERVE command issued. Close: no RELEASE. |
| force + retain + single | Open: normal except SCIORESET issued prior to any SCSI commands. Close: no RELEASE. |
| force + single | Open: normal except SCIORESETissued prior to any SCSI commands. Close: normal. |
| force + single + no_reserve | Open: normal except SCIORESET issued prior to any SCSI commands. No RESERVE command issued. Close: no RELEASE. |
| no_reserve | Open: no RESERVE. Close: no RELEASE. |
| retain | Open: normal. Close: no RELEASE. |
| retain + no_reserve | Open: no RESERVE. Close: no RELEASE. |
| retain + single | Open: normal. Close: no RELEASE. |
| retain + single + no_reserve | Open: normal except no RESERVE command issued. Close: no RELEASE. |
| single | Open: normal. Close: normal. |
| single + no_reserve | Open: no RESERVE. Close: no RELEASE. |

## Closing the SCSI Device

When a SCSI device driver is preparing to close a device through the SCSI adapter device driver, it must ensure that all transactions are complete.

When the SCSI adapter device driver receives a **SCIOSTOP** ioctl operation and there are pending I/O requests, the ioctl operation does not return until all have completed. New requests received during this time are rejected from the adapter device driver's **ddstrategy** routine.

When the SCSI adapter device driver receives an **SCIOSTOPTGT** ioctl operation, it must forcibly free any receive data buffers that have been queued to the SCSI device driver for this device and have not been returned to the SCSI adapter device driver through the buffer free routine. The SCSI device driver is responsible for making sure all the receive data buffers are freed before calling the **SCIOSTOPTGT** ioctl operation. However, the SCSI adapter device driver must check that this is done, and, if necessary, forcibly free the buffers. The buffers must be freed because those not freed result in memory areas being permanently lost to the system (until the next boot).

To allow the SCSI adapter device driver to free buffers that are sent to the SCSI device driver but never returned, it must track which **tm_bufs** are currently queued to the SCSI device driver. Tracking **tm_bufs** requires the SCSI adapter device driver to violate the general SCSI rule, which states the SCSI adapter device driver should not modify the **tm_bufs** structure while it is queued to the SCSI device driver. This exception to the rule is necessary because it is never acceptable not to free memory allocated from the system.

## SCSI Error Processing

It is the responsibility of the SCSI device driver to process SCSI check conditions and other returned errors properly.

The SCSI adapter device driver only passes SCSI commands without otherwise processing them and is not responsible for device error recovery.

## Device Driver and Adapter Device Driver Interfaces

The SCSI device drivers can have both character (raw) and block special files in the **/dev** directory.

The SCSI adapter device driver has only character (raw) special files in the **/dev** directory and has only the **ddconfig**, **ddopen**, **ddclose**, **dddump**, and **ddioctl** entry points available to operating system programs. The **ddread** and **ddwrite** entry points are not implemented.

Internally, the **devsw** table has entry points for the **ddconfig**, **ddopen**, **ddclose**, **dddump**, **ddioctl**, and **ddstrategy** routines. The SCSI device drivers pass their SCSI commands to the SCSI adapter device driver by calling the SCSI adapter device driver **ddstrategy** routine. (This routine is unavailable to other operating system programs due to the lack of a block-device special file.)

Access to the SCSI adapter device driver's **ddconfig**, **ddopen**, **ddclose**, **dddump**, **ddioctl**, and **ddstrategy** entry points by the SCSI device drivers is performed through the kernel services provided. These include such services as **fp_opendev**, **fp_close**, **fp_ioctl**, **devdump**, and **devstrategy**.

## Performing SCSI Dumps

A SCSI adapter device driver must have a **dddump** entry point if it is used to access a system dump device. A SCSI device driver must have a **dddump** entry point if it drives a dump device. Examples of dump devices are disks and tapes.

**Note:** SCSI adapter-device-driver writers should be aware that system services providing interrupt and timer services are unavailable for use in the dump routine. Kernel DMA services are assumed to be available for use by the dump routine. The SCSI adapter device driver should be designed to ignore extra **DUMPINIT** and **DUMPSTART** commands to the **dddump** entry point.

The **DUMPQUERY** option should return a minimum transfer size of 0 bytes, and a maximum transfer size equal to the maximum transfer size supported by the SCSI adapter device driver.

Calls to the SCSI adapter device driver **DUMPWRITE** option should use the *arg* parameter as a pointer to the **sc_buf** structure to be processed. Using this interface, a SCSI **write** command can be run on a previously started (opened) target device. The *uiop* parameter is ignored by the SCSI adapter device driver during the **DUMPWRITE** command. Spanned, or consolidated, commands are not supported using the **DUMPWRITE** option. Gathered **write** commands are also not supported using the **DUMPWRITE** option. No queuing of **sc_buf** structures is supported during dump processing because the dump routine runs essentially as a subroutine call from the caller's dump routine. Control is returned when the entire **sc_buf** structure has been processed.

> **Attention:** Also, both adapter-device-driver and device-driver writers should be aware that any error occurring during the **DUMPWRITE** option is considered unsuccessful. Therefore, no error recovery is employed during the **DUMPWRITE**. Return values from the call to the **dddump** routine indicate the specific nature of the failure.

Successful completion of the selected operation is indicated by a 0 return value to the subroutine. Unsuccessful completion is indicated by a return code set to one of the following values for the **errno** global variable. The various **sc_buf** status fields, including the `b_error` field, are not set by the SCSI

adapter device driver at completion of the **DUMPWRITE** command. Error logging is, of necessity, not supported during the dump.

- An **errno** value of **EINVAL** indicates that a request that was not valid passed to the SCSI adapter device driver, such as to attempt a **DUMPSTART** command before successfully executing a **DUMPINIT** command.
- An **errno** value of **EIO** indicates that the SCSI adapter device driver was unable to complete the command due to a lack of required resources or an I/O error.
- An **errno** value of **ETIMEDOUT** indicates that the adapter did not respond with completion status before the passed command time-out value expired.

## SCSI Target-Mode Overview

The SCSI target-mode interface is intended to be used with the SCSI initiator-mode interface to provide the equivalent of a full-duplex communications path between processor type devices. Both communicating devices must support target-mode and initiator-mode.

**Note:** This operation is not supported by all SCSI I/O controllers.

To work with the SCSI subsystem in this manner, an attached device's target-mode and initiator-mode interfaces must meet certain minimum requirements:

- The device's target-mode interface must be capable of receiving and processing at least the following SCSI commands:
  - **send**
  - **request sense**
  - **inquiry**

  The data returned by the **inquiry** command must set the peripheral device type field to processor device. The device should support the vendor and product identification fields. Additional functional SCSI requirements, such as SCSI message support, must be addressed by examining the detailed functional specification of the SCSI initiator that the target-mode device is attached to.
- The attached device's initiator mode interface must be capable of sending the following SCSI commands:
  - **send**
  - **request sense**

  In addition, the **inquiry** command should be supported by the attached initiator if it needs to identify SCSI target devices. Additional functional SCSI requirements, such as SCSI message support, must be addressed by examining the detailed functional specification of the SCSI target that the initiator-mode device is attached to.

### Configuring and Using SCSI Target Mode

The adapter, acting as either a target or initiator device, requires its own SCSI ID. This ID, as well as the IDs of all attached devices on this SCSI bus, must be unique and between 0 and 7, inclusive.

Because each device on the bus must be at a unique ID, the user must complete any installation and configuration of the SCSI devices required to set the correct IDs before physically cabling the devices together. Failure to do so will produce unpredictable results.

SCSI target mode in the SCSI subsystem does not attempt to implement any receive-data protocol, with the exception of actions taken to prevent an application from excessive receive-data-buffer usage. Any protocol required to maintain or otherwise manage the communications of data must be implemented in user-supplied programs. The only delays in receiving data are those inherent in the SCSI subsystem and the hardware environment in which it operates.

The SCSI target mode is capable of simultaneously receiving data from all attached SCSI IDs using SCSI **send** commands. In target-mode, the host adapter is assumed to act as a single SCSI Logical Unit Number

(LUN) at its assigned SCSI ID. Therefore, only one logical connection is possible between each attached SCSI initiator on the SCSI Bus and the host adapter. The SCSI subsystem is designed to be fully capable of simultaneously sending SCSI commands in initiator-mode while receiving data in target-mode.

## Managing Receive-Data Buffers

In the SCSI subsystem target-mode interface, the SCSI adapter device driver is responsible for managing the receive-data buffers versus the SCSI device driver because the buffering is dependent upon how the adapter works.

It is not possible for the SCSI device driver to run a single approach that is capable of making full use of the performance advantages of various adapters' buffering schemes. With the SCSI adapter device driver layer performing the buffer management, the SCSI device driver can be interfaced to a variety of adapter types and can potentially get the best possible performance out of each adapter. This approach also allows multiple SCSI target-mode device drivers to be run on top of adapters that use a shared-pool buffer management scheme. This would not be possible if the target-mode device drivers managed the buffers.

## Understanding Target-Mode Data Pacing

Because it is possible for the attached initiator device to send data faster than the host operating system and associated application can process it, eventually the situation arises in which all buffers for this device instance are in use at the same time.

There are two possible scenarios:

- The previous **send** command has been received by the adapter, but there is no space for the next **send** command.
- The **send** command is not yet completed, and there is no space for the remaining data.

In both cases, the combination of the SCSI adapter device driver and the SCSI adapter must be capable of stopping the flow of data from the initiator device.

### *SCSI Adapter Device Driver*

The adapter can handle both cases described previously by simply accepting the **send** command (if newly received) and then disconnecting during the data phase. When buffer space becomes available, the SCSI adapter reconnects and continues the data transfer. As an alternative, when handling a newly received command, a check condition can be given back to the initiator to indicate a lack of resources. The implementation of this alternative is adapter-dependent. The technique of accepting the command and then disconnecting until buffer space is available should result in better throughput, as it avoids both a **request sense** command and the retry of the **send** command.

For adapters allowing a shared pool of buffers to be used for all attached initiators' data transfers, an additional problem can result. If any single initiator instance is allowed to transfer data continually, the entire shared pool of buffers can fill up. These filled-up buffers prevent other initiator instances from transferring data. To solve this problem, the combination of the SCSI adapter device driver and the host SCSI adapter must stop the flow of data from a particular initiator ID on the bus. This could include disconnecting during the data phase for a particular ID but allowing other IDs to continue data transfer. This could begin when the number of **tm_buf** structures on a target-mode instance's **tm_buf** queue equals the number of buffers allocated for this device. When a threshold percentage of the number of buffers is processed and returned to the SCSI adapter device driver's buffer-free routine, the ID can be enabled again for the continuation of data transfer.

### *SCSI Device Driver*

The SCSI device driver can optionally be informed by the SCSI adapter device driver whenever all buffers for this device are in use. This is known as a maximum-buffer-usage event.

To pass this information, the SCSI device driver must be registered for notification of asynchronous event status from the SCSI adapter device driver. Registration is done by calling the SCSI adapter device-driver ioctl entry point with the **SCIOEVENT** operation. If registering for event notification, the SCSI device driver receives notification of all asynchronous events, not just the maximum buffer usage event.

## Understanding the SCSI Target Mode Device Driver Receive Buffer Routine

The SCSI target-mode device-driver **receive buffer** routine must be a pinned routine that the SCSI adapter device driver can directly address. This routine is called directly from the SCSI adapter device driver hardware interrupt handling routine. The SCSI device driver writer must be aware of how this routine affects the design of the SCSI device driver.

First, because the **receive buffer** routine is running on the hardware interrupt level, the SCSI device driver must limit operations in order to limit routine processing time. In particular, the data copy, which occurs because the data is queued ahead of the user read request, must not occur in the **receive buffer** routine. Data copying in this routine will adversely affect system response time. Data copy is best performed in a process level SCSI device-driver routine. This routine sleeps, waiting for data, and is awakened by the **receive buffer** routine. Typically, this process level routine is the SCSI device driver's **read** routine.

Second, the **receive buffer** routine is called at the SCSI adapter device driver hardware interrupt level, so care must be taken when disabling interrupts. They must be disabled to the correct level in places in the SCSI device driver's lower run priority routines, which manipulate variables also modified in the **receive buffer** routine. To allow the SCSI device driver to disable to the correct level, the SCSI adapter device-driver writer must provide a configuration database attribute, named **intr_priority**, that defines the interrupt class, or priority, that the adapter runs on. The SCSI device-driver configuration method should pass this attribute to the SCSI device driver along with other configuration data for the device instance.

Third, the SCSI device-driver writer must follow any other general system rules for writing a routine that must run in an interrupt environment. For example, the routine must not attempt to sleep or wait on I/O operations. It can perform wake-up calls to allow the process level to handle those operations.

Duties of the SCSI device driver **receive buffer** routine include:

- Matching the data with the appropriate target-mode instance.
- Queuing the **tm_buf** structures to the appropriate target-mode instance.
- Waking up the process-level routine for further processing of the received data.

After the **tm_buf** structure has been passed to the SCSI device driver **receive buffer** routine, the SCSI device driver is considered to be responsible for it. Responsibilities include processing the data and any error conditions and also maintaining the `next` pointer for chained **tm_buf** structures. The SCSI device driver's responsibilities for the **tm_buf** structures end when it passes the structure back to the SCSI adapter device driver.

Until the **tm_buf** structure is again passed to the SCSI device driver **receive buffer** routine, the SCSI adapter device driver is considered responsible for it. The SCSI adapter device-driver writer must be aware that during the time the SCSI device driver is responsible for the **tm_buf** structure, it is still possible for the SCSI adapter device driver to access the structure's contents. Access is possible because only one copy of the structure is in memory, and only a pointer to the structure is passed to the SCSI device driver.

**Note:** Under no circumstances should the SCSI adapter device driver access the structure or modify its contents while the SCSI device driver is responsible for it, or the other way around.

It is recommended that the SCSI device-driver writer implement a threshold level to wake up the process level with available **tm_buf** structures. This way, processing for some of the buffers, including copying the data to the user buffer, can be overlapped with time spent waiting for more data. It is also recommended the writer implement a threshold level for these buffers to handle cases where the **send** command data length exceeds the aggregate receive-data buffer space. A suggested threshold level is 25% of the device's total buffers. That is, when 25% or more of the number of buffers allocated for this device is queued and no end to the **send** command is encountered, the SCSI device driver receive buffer routine should wake the process level to process these buffers.

## Understanding the tm_buf Structure

The **tm_buf** structure is used for communication between the SCSI device driver and the SCSI adapter device driver for a target-mode received-data buffer. The **tm_buf** structure is passed by pointer directly to

routines whose entry points have been registered through the **SCIOSTARTTGT** ioctl operation of the SCSI adapter device driver.

The SCSI device driver is required to call this ioctl operation when opening a target-mode device instance.

### *Fields in the tm_buf Structure*

The **tm_buf** structure contains certain fields used to pass a received data buffer from the SCSI adapter device driver to the SCSI device driver. Other fields are used to pass returned status back to the SCSI device driver. After processing the data, the **tm_buf** structure is passed back from the SCSI device driver to the SCSI adapter device driver to allow the buffer to be reused.

The **tm_buf** structure is defined in the **/usr/include/sys/scsi.h** file and contains the following fields:

**Note:** Reserved fields must not be modified by the SCSI device driver, unless noted otherwise. Nonreserved fields can be modified, except where noted otherwise.

1. The `tm_correlator` field is an optional field for the SCSI device driver. This field is a copy of the field with the same name that was passed by the SCSI device driver in the **SCIOSTARTTGT** ioctl. The SCSI device driver should use this field to speed the search for the target-mode device instance the **tm_buf** structure is associated with. Alternatively, the SCSI device driver can combine the `tm_buf.user_id` and `tm_buf.adap_devno` fields to find the associated device.
2. The `adap_devno` field is the device major and minor numbers of the adapter instance on which this target mode device is defined. This field can be used to find the particular target-mode instance the **tm_buf** structure is associated with.

   **Note:** The SCSI device driver must not modify this field.
3. The `data_addr` field is the kernel space address where the data begins for this buffer.
4. The `data_len` field is the length of valid data in the buffer starting at the **tm_buf.data_addr** location in memory.
5. The `user_flag` field is a set of bit flags that can be set to communicate information about this data buffer to the SCSI device driver. Except where noted, one or more of the following flags can be set:

   **TM_HASDATA**
   Set to indicate a valid **tm_buf** structure

   **TM_MORE_DATA**
   Set if more data is coming (that is, more **tm_buf** structures) for a particular **send** command. This is only possible for adapters that support spanning the **send** command data across multiple receive buffers. This flag cannot be used with the **TM_ERROR** flag.

   **TM_ERROR**
   Set if any error occurred on a particular **send** command. This flag cannot be used with the **TM_MORE_DATA** flag.
6. The `user_id` field is set to the SCSI ID of the initiator that sent the data to this target mode instance. If more than one adapter is used for target mode in this system, this ID might not be unique. Therefore, this field must be used in combination with the `tm_buf.adap_devno` field to find the target-mode instance this ID is associated with.

   **Note:** The SCSI device driver must not modify this field.
7. The `status_validity` field contains the following bit flag:

   **SC_ADAPTER_ERROR**
   Indicates the `tm_buf.general_card_status` is valid.
8. The `general_card_status` field is a returned status field that gives a broad indication of the class of error encountered by the adapter. This field is valid when its status-validity bit is set in the `tm_buf.status_validity` field. The definition of this field is the same as that found in the **sc_buf** structure definition, except the **SC_CMD_TIMEOUT** value is not possible and is never returned for a target-mode transfer.
9. The `next` field is a **tm_buf** pointer that is either null, meaning this is the only or last **tm_buf** structure, or else contains a non-null pointer to the next **tm_buf** structure.

## Understanding the Running of SCSI Target-Mode Requests

The target-mode interface provided by the SCSI subsystem is designed to handle data reception from SCSI **send** commands.

The host SCSI adapter acts as a secondary device that waits for an attached initiator device to issue a SCSI **send** command. The SCSI **send** command data is received by buffers managed by the SCSI adapter device driver. The **tm_buf** structure is used to manage individual buffers. For each buffer of data received from an attached initiator, the SCSI adapter device driver passes a **tm_buf** structure to the SCSI device driver for processing. Multiple **tm_buf** structures can be linked together and passed to the SCSI device driver at one time. When the SCSI device driver has processed one or more **tm_buf** structures, it passes the **tm_buf** structures back to the SCSI adapter device driver so they can be reused.

### *Detailed Running of Target-Mode Requests*

When a **send** command is received by the host SCSI adapter, data is placed in one or more receive-data buffers. These buffers are made available to the adapter by the SCSI adapter device driver. The procedure by which the data gets from the SCSI bus to the system-memory buffer is adapter-dependent.

The SCSI adapter device driver takes the received data and updates the information in one or more **tm_buf** structures in order to identify the data to the SCSI device driver. This process includes filling the `tm_correlator`, `adap_devno`, `data_addr`, `data_len`, `user_flag`, and `user_id` fields. Error status information is put in the `status_validity` and `general_card_status` fields. The `next` field is set to null to indicate this is the only element, or set to non-null to link multiple **tm_buf** structures. If there are multiple **tm_buf** structures, the final `tm_buf.next` field is set to null to end the chain. If there are multiple **tm_buf** structures and they are linked, they must all be from the same initiator SCSI ID. The `tm_buf.tm_correlator` field, in this case, has the same value as it does in the **SCIOSTARTTGT** ioctl operation to the SCSI adapter device driver. The SCSI device driver should use this field to speed the search for the target-mode instance designated by this **tm_buf** structure. For example, when using the value of `tm_buf.tm_correlator` as a pointer to the device-information structure associated with this target-mode instance.

Each **send** command, no matter how short its data length, requires its own **tm_buf** structure. For host SCSI adapters capable of spanning multiple receive-data buffers with data from a single **send** command, the SCSI adapter device driver must set the **TM_MORE_DATA** flag in the `tm_buf.user_flag` fields of all but the final **tm_buf** structure holding data for the **send** command. The SCSI device driver must be designed to support the **TM_MORE_DATA** flag. Using this flag, the target-mode SCSI device driver can associate multiple buffers with the single transfer they represent. The end of a **send** command will be the boundary used by the SCSI device driver to satisfy a user read request.

The SCSI adapter device driver is responsible for sending the **tm_buf** structures for a particular initiator SCSI ID to the SCSI device driver in the order they were received. The SCSI device driver is responsible for processing these **tm_buf** structures in the order they were received. There is no particular ordering implied in the processing of simultaneous **send** commands from different SCSI IDs, as long as the data from an individual SCSI ID's **send** command is processed in the order it was received.

The pointer to the **tm_buf** structure chain is passed by the SCSI adapter device driver to the SCSI device driver's receive buffer routine. The address of this routine is registered with the SCSI adapter device driver by the SCSI device driver using the **SCIOSTARTTGT** ioctl. The duties of the receive buffer routine include queuing the **tm_buf** structures and waking up a process-level routine (typically the SCSI device driver's **read** routine) to process the received data.

When the process-level SCSI device driver routine finishes processing one or more **tm_buf** structures, it passes them to the SCSI adapter device driver's buffer-free routine. The address of this routine is registered with the SCSI device driver in an output field in the structure passed to the SCSI adapter device driver **SCIOSTARTTGT** ioctl operation. The buffer-free routine must be a pinned routine the SCSI device driver can directly access. The buffer-free routine is typically called directly from the SCSI device driver buffer-handling routine. The SCSI device driver chains one or more **tm_buf** structures by using the `next` field (a null value for the last tm_buf next field ends the chain). It then passes a pointer, which points to the head of the chain, to the SCSI adapter device driver buffer-free routine. These **tm_buf** structures must all be for the same target-mode instance. Also, the SCSI device driver must not modify the `tm_buf.user_id` or `tm_buf.adap_devno` field.

The SCSI adapter device driver takes the **tm_buf** structures passed to its buffer-free routine and attempts to make the described receive buffers available to the adapter for future data transfers. Because it is desirable to keep as many buffers as possible available to the adapter, the SCSI device driver should pass processed **tm_buf** structures to the SCSI-adapter device driver's buffer-free routine as quickly as possible. The writer of a SCSI device driver should avoid requiring the last buffer of a **send** command to be received before processing buffers, as this could cause a situation where all buffers are in use and the **send** command has not completed. It is recommended that the writer therefore place a threshold of 25% on the free buffers. That is, when 25% or more of the number of buffers allocated for this device have been processed and the **send** command is not completed, the SCSI device driver should free the processed buffers by passing them to the SCSI adapter device driver's buffer-free routine.

# Required SCSI Adapter Device Driver ioctl Commands

Various ioctl operations must be performed for proper operation of the SCSI adapter device driver. The ioctl operations described here are the minimum set of commands the SCSI adapter device driver must implement to support SCSI device drivers.

Other operations might be required in the SCSI adapter device driver to support, for example, system management facilities and diagnostics. SCSI device driver writers also need to understand these ioctl operations.

Every SCSI adapter device driver must support the **IOCINFO** ioctl operation. The structure to be returned to the caller is the **devinfo** structure, including the **scsi** union definition for the SCSI adapter, which can be found in the **/usr/include/sys/devinfo.h** file. The SCSI device driver should request the **IOCINFO** ioctl operation (probably during its open routine) to get the maximum transfer size of the adapter.

**Note:** The SCSI adapter device driver ioctl operations can only be called from the process level. They cannot be run from a call on any more favored priority levels. Attempting to call them from a more favored priority level can result in a system crash.

## Initiator-Mode ioctl Commands

The following **SCIOSTART** and **SCIOSTOP** operations must be sent by the SCSI device driver (for the open and close routines, respectively) for each device. They cause the SCSI adapter device driver to allocate and initialize internal resources.

The **SCIOHALT** ioctl operation is used to abort pending or running commands, usually after signal processing by the SCSI device driver. This might be used by a SCSI device driver to end an operation instead of waiting for completion or a time out. The **SCIORESET** operation is provided for clearing device hard errors and competing initiator reservations during open processing by the SCSI device driver. The **SCIOGTHW** operation is supported by SCSI adapter device drivers that support gathered write commands to target devices.

Except where noted otherwise, the *arg* parameter for each of the ioctl operations described here must contain a long integer. In this field, the least significant byte is the SCSI LUN and the next least significant byte is the SCSI ID value. (The upper two bytes are reserved and should be set to 0.) This provides the information required to allocate or deallocate resources and perform SCSI bus operations for the ioctl operation requested.

The following information is provided on the various ioctl operations:

**SCIOSTART**
This operation allocates and initializes SCSI device-dependent information local to the SCSI adapter device driver. Run this operation only on the first open of an ID/LUN device. Subsequent **SCIOSTART** commands to the same ID/LUN fail unless an intervening **SCIOSTOP** command is issued.

The following values for the **errno** global variable are supported:

**0**
Indicates successful completion.

**EIO**
Indicates lack of resources or other error-preventing device allocation.

**EINVAL**
Indicates that the selected SCSI ID and LUN are already in use, or the SCSI ID matches the adapter ID.

**ETIMEDOUT**
Indicates that the command did not complete.

**SCIOSTOP**
This operation deallocates resources local to the SCSI adapter device driver for this SCSI device. This should be run on the last close of an ID/LUN device. If an **SCIOSTART** operation has not been previously issued, this command is unsuccessful.

The following values for the **errno** global variable should be supported:

**0**
Indicates successful completion.

**EIO**
Indicates error preventing device deallocation.

**EINVAL**
Indicates that the selected SCSI ID and LUN have not been started.

**ETIMEDOUT**
Indicates that the command did not complete.

**SCIOCMD**
The SCIOCMD SCSI Adapter Device Driver ioctl Operation provides the means for issuing any SCSI command to the specified device after the SCSI device has been successfully started (SCIOSTART). The SCSI adapter driver performs no error recovery other then issuing a request sense for a SCSI check condition error. If the caller allocated an autosense buffer, then the request sense data is returned in that buffer. The SCSI adapter driver will not log any errors in the system error log for failures on a SCIOCMD operation. The following is a typical call:

```
rc = ioctl(adapter, SCIOCMD, &iocmd);
```

where *adapter* is a file descriptor and iocmd is an **sc_passthru** structure as defined in the **/usr/include/sys/scsi.h** header file. The SCSI ID and LUN should be placed in the **sc_passthru** parameter block.

The SCSI status byte and the adapter status bytes are returned through the **sc_passthru** structure. If the SCIOCMD operation returns a value of -1 and the errno global variable is set to a nonzero value, the requested operation has failed. In this case, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken.

If a SCIOCMD operation fails because a field in the **sc_passthru** structure has an invalid value, then the subroutine will return a value of -1 and set the errno global variable to EINVAL. In addition the **einval_arg** field will be set to the field number (starting with 1 for the version field) of the field that had an invalid value. A value of 0 for the **einval_arg** field indicates no additional information on the failure is available.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum, a value of -1 is returned and the errno global variable set to a value of EINVAL. Refer to the Small Computer System Interface (SCSI) Specification for the applicable device to get request sense information.

Possible errno values are:

**EIO**
A system error has occurred. Consider retrying the operation several (three or more) times, because another attempt might be successful. If an EIO error occurs and the **status_validity** field is set to SC_SCSI_ERROR, then the **scsi_status** field has a valid value and should be inspected.

If the **status_validity** field is zero and remains so on successive retries, then an unrecoverable error has occurred with the device.

If the **status_validity** field is SC_SCSI_ERROR and the **scsi_status** field contains a Check Condition status, then a SCSI request sense should be issued using the SCIOCMD ioctl to recover the the sense data.

**EFAULT**
A user process copy has failed.

**EINVAL**
The device is not opened or the caller has set a field in the **sc_passthru** structure to an invalid value.

**EACCES**
The adapter is in diagnostics mode.

**ENOMEM**
A memory request has failed.

**ETIMEDOUT**
The command has timed out, which indicates the operation did not complete before the time-out value was exceeded. Consider retrying the operation.

**ENODEV**
The device is not responding.

**Note:** This operation requires the **SCIOSTART** operation to be run first.

If the FCP **SCIOCMD** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SC_RET_ID** flag set. This field is set only if the **world_wide_name** and **node_ name** fields were provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field will contain the new **scsi_id** value.

The `version` field of the `scsi_passthru` structure can be set to the value of `SC_VERSION_2` in `/usr/include/sys/scsi.h` or `SCSI_VERSION_2` in `/usrinclude/sys/scsi_buf.h`, and the user can provide the following fields:

- `variable_cdb_ptr` - pointer to a buffer that contains the SCSI *cdb* variable.
- `variable_cdb_length` - the length of the variable *cdb* to which the *variable_cdb_ptr* points.

When the `SCIOCMD` ioctl request with the `version` field set to `SCSI_VERSION_2` completes and the device did not fully satisfy the request, the `residual` field indicates left over data. If the request completes successfully, the `residual` field indicates the device does not have all the requested data. If the request did not complete successfully, check the `status_validity` to see whether a valid SCSI bus problem exists. If a valid SCSI bus problem exists, the `residual` field indicates the number of bytes by which the device failed to complete the request.

**SCIOHALT**
This operation halts outstanding transactions to this ID/LUN device and causes the SCSI adapter device driver to stop accepting transactions for this device. This situation remains in effect until the SCSI device driver sends another transaction with the **SC_RESUME** flag set (in the **sc_buf.flags** field) for this ID/LUN combination. The **SCIOHALT** ioctl operation causes the SCSI adapter device driver to fail the command in progress, if any, as well as all queued commands for the device with a return value of **ENXIO** in the `sc_buf.bufstruct.b_error` field. If an **SCIOSTART** operation has not been previously issued, this command fails.

The following values for the **errno** global variable are supported:

**0**
Indicates successful completion.

**EIO**
Indicates an unrecovered I/O error occurred.

**EINVAL**
Indicates that the selected SCSI ID and LUN have not been started.

**ETIMEDOUT**
Indicates that the command did not complete.

**SCIORESET**
This operation causes the SCSI adapter device driver to send a SCSI Bus Device Reset (BDR) message to the selected SCSI ID. For this operation, the SCSI device driver should set the LUN in the *arg* parameter to the LUN ID of a LUN on this SCSI ID, which has been successfully started using the **SCIOSTART** operation.

The SCSI device driver should use this command only when directed to do a *forced open*. This occurs in two possible situations: one, when it is desirable to force the device to drop a SCSI reservation; two, when the device needs to be reset to clear an error condition (for example, when running diagnostics on this device).

**Note:** In normal system operation, this command should not be issued, as it would force the device to drop a SCSI reservation another initiator (and, hence, another system) might have. If an **SCIOSTART** operation has not been previously issued, this command is unsuccessful.

The following values for the **errno** global variable are supported:

**0**
Indicates successful completion.

**EIO**
Indicates an unrecovered I/O error occurred.

**EINVAL**
Indicates that the selected SCSI ID and LUN have not been started.

**ETIMEDOUT**
Indicates that the command did not complete.

**SCIOGTHW**
This operation is only supported by SCSI adapter device drivers that support gathered write commands. The purpose of the operation is to indicate support for gathered writes to SCSI device drivers that intend to use this facility. If the SCSI adapter device driver does not support gathered write commands, it must fail the operation. The SCSI device driver should call this operation from its open routine for a particular device instance. If the operation is unsuccessful, the SCSI device driver should not attempt to run a gathered write command.

The *arg* parameter to the **SCIOGTHW** is set to null by the caller to indicate that no input parameter is passed:

The following values for the **errno** global variable are supported:

**0**
Indicates successful completion and in particular that the adapter driver supports gathered writes.

**EINVAL**
Indicates that the SCSI adapter device driver does not support gathered writes.

## Target-Mode ioctl Commands

The following **SCIOSTARTTGT** and **SCIOSTOPTGT** operations must be sent by the SCSI device driver (for the open and close routines, respectively) for each target-mode device instance. This causes the SCSI adapter device driver to allocate and initialize internal resources, and, if necessary, prepare the hardware for operation.

Target-mode support in the SCSI device driver and SCSI adapter device driver is optional. A failing return code from these commands, in the absence of any programming error, indicates target mode is not supported. If the SCSI device driver requires target mode, it must check the return code to verify the SCSI adapter device driver supports it.

Only a kernel process or device driver can call these ioctls. If attempted by a user process, the ioctl will fail, and the **errno** global variable will be set to **EPERM**.

The following information is provided on the various target-mode ioctl operations:

**SCIOSTARTTGT**

This operation opens a logical path to a SCSI initiator device. It allocates and initializes SCSI device-dependent information local to the SCSI adapter device driver. This is run by the SCSI device driver in its open routine. Subsequent **SCIOSTARTTGT** commands to the same ID (LUN is always 0) are unsuccessful unless an intervening **SCIOSTOPTGT** is issued. This command also causes the SCSI adapter device driver to allocate system buffer areas to hold data received from the initiator, and makes the adapter ready to receive data from the selected initiator.

The *arg* parameter to the **SCIOSTARTTGT** should be set to the address of an **sc_strt_tgt** structure, which is defined in the **/usr/include/sys/scsi.h** file. The following parameters are supported:

***id***
The caller fills in the SCSI ID of the attached SCSI initiator.

***lun***
The caller sets the LUN to 0, as the initiator LUN is ignored for received data.

***buf_size***
The caller specifies size in bytes to be used for each receive buffer allocated for this host target instance.

***num_bufs***
The caller specifies how many buffers to allocate for this target instance.

***tm_correlator***
The caller optionally places a value in this field to be passed back in each **tm_buf** for this target instance.

***recv_func***
The caller places in this field the address of a pinned routine the SCSI adapter device driver should call to pass **tm_bufs** received for this target instance.

***free_func***
This is an output parameter the SCSI adapter device driver fills with the address of a pinned routine that the SCSI device driver calls to pass **tm_bufs** after they have been processed. The SCSI adapter device driver ignores the value passed as input.

**Note:** All reserved fields should be set to 0 by the caller.

The following values for the **errno** global variable are supported:

**0**
Indicates successful completion.

**EINVAL**
An **SCIOSTARTTGT** command has already been issued to this SCSI ID.

The passed SCSI ID is the same as that of the adapter.

The LUN ID field is not set to zero.

The *buf_size* is not valid. This is an adapter dependent value.

The *Num_bufs* is not valid. This is an adapter dependent value.

The *recv_func* value, which cannot be null, is not valid.

**EPERM**
Indicates the caller is not running in kernel mode, which is the only mode allowed to run this operation.

**ENOMEM**
Indicates that a memory allocation failure has occurred.

**EIO**
Indicates an I/O error occurred, preventing the device driver from completing **SCIOSTARTTGT** processing.

**SCIOSTOPTGT**
This operation closes a logical path to a SCSI initiator device. It causes the SCSI adapter device driver to deallocate device dependent information areas allocated in response to a **SCIOSTARTTGT** operation. It also causes the SCSI adapter device driver to deallocate system buffer areas used to hold data received from the initiator, and to disable the host adapter's ability to receive data from the selected initiator.

The *arg* parameter to the **SCIOSTOPTGT** ioctl should be set to the address of an **sc_stop_tgt** structure, which is defined in the **/usr/include/sys/scsi.h** file. The caller fills in the **id** field with the SCSI ID of the SCSI initiator, and sets the **lun** field to 0 as the initiator LUN is ignored for received data. Reserved fields should be set to 0 by the caller.

The following values for the **errno** global variable should be supported:

**0**
Indicates successful completion.

**EINVAL**
An **SCIOSTARTTGT** command has not been previously issued to this SCSI ID.

**EPERM**
Indicates the caller is not running in kernel mode, which is the only mode allowed to run this operation.

## Target- and Initiator-Mode ioctl Commands

For either target or initiator mode, the SCSI device driver can issue an **SCIOEVENT** ioctl operation to register for receiving asynchronous event status from the SCSI adapter device driver for a particular device instance. This is an optional call for the SCSI device driver, and is optionally supported for the SCSI adapter device driver. A failing return code from this command, in the absence of any programming error, indicates it is not supported. If the SCSI device driver requires this function, it must check the return code to verify the SCSI adapter device driver supports it.

Only a kernel process or device driver can invoke these ioctls. If attempted by a user process, the ioctl will fail, and the **errno** global variable will be set to **EPERM**.

The event registration performed by this ioctl operation is allowed once per device session. Only the first **SCIOEVENT** ioctl operation is accepted after the device session is opened. Succeeding **SCIOEVENT** ioctl operations will fail, and the **errno** global variable will be set to **EINVAL**. The event registration is canceled automatically when the device session is closed.

The *arg* parameter to the **SCIOEVENT** ioctl operation should be set to the address of an **sc_event_struct** structure, which is defined in the **/usr/include/sys/scsi.h** file. The following parameters are supported:

| Item | Description |
|---|---|
| *id* | The caller sets *id* to the SCSI ID of the attached SCSI target device for initiator-mode. For target-mode, the caller sets the *id* to the SCSI ID of the attached SCSI initiator device. |
| *lun* | The caller sets the *lun* field to the SCSI LUN of the attached SCSI target device for initiator-mode. For target-mode, the caller sets the *lun* field to 0. |
| *mode* | Identifies whether the initiator- or target-mode device is being registered. These values are possible:<br>**SC_IM_MODE**<br>This is an initiator mode device.<br>**SC_TM_MODE**<br>This is a target mode device. |

| Item | Description |
| --- | --- |
| *async_correlator* | The caller places a value in this optional field, which is saved by the SCSI adapter device driver and returned when an event occurs in this field in the **sc_event_info** structure. This structure is defined in the **/user/include/sys/scsi.h** file. |
| *async_func* | The caller fills in the address of a pinned routine that the SCSI adapter device driver calls whenever asynchronous event status is available. The SCSI adapter device driver passes a pointer to a **sc_event_info** structure to the caller's **async_func** routine. |

**Note:** All reserved fields should be set to 0 by the caller.

The following values for the **errno** global variable are supported:

| Item | Description |
| --- | --- |
| **0** | Indicates successful completion. |
| **EINVAL** | Either an **SCIOSTART** or **SCIOSTARTTGT** has not been issued to this device instance, or this device is already registered for async events. |
| **EPERM** | Indicates the caller is not running in kernel mode, which is the only mode allowed to run this operation. |

# SCSI Architectural Model Subsystem

This overview describes the interface between a SCSI Architectural Model (SAM) device driver and a SAM adapter device driver. *SAM* is a set of multiple physical transport types, all of which make use of the SCSI command set.

You can use the following physical transport types in SAM:

- Fibre Channel Protocol for SCSI (FCP)
- iSCSI
- Serial Attached SCSI (SAS)

For information about the traditional parallel bus implementation of SCSI, see Small Computer System Interface Subsystem (Parallel SCSI).

The SAM subsystem is directed toward those wishing to design and write a SAM storage device driver that interfaces with an existing SAM adapter device driver. It is also meant for those wishing to design and write a SAM adapter device driver that interfaces with existing SAM storage device drivers.

**Related concepts**

Logical File System Kernel Services
The Logical File System services (also known as the **fp_**services) allow processes running in kernel mode to open and manipulate files in the same way that user-mode processes do. Data access limitations make it unreasonable to accomplish these tasks with system calls, so a subset of the file system calls has been provided with an alternate kernel-only interface.

**Related information**

scdisk SCSI Device Driver

## Programming SAM Device Drivers

The SAM subsystem has two parts.

- Device Driver
- Adapter Device Driver

The adapter device driver is designed to shield you from having to communicate directly with the system I/O hardware. This gives you the ability to successfully write a device driver without having a detailed knowledge of the system hardware. You can look at the subsystem as a two-tiered structure in which the adapter device driver is the bottom or supporting layer. As a programmer, you need only worry about the upper layer. This chapter only discusses writing a device driver, because the adapter device driver is already provided.

The adapter device driver, or lower layer, is responsible only for the communications to and from the bus, and any error logging and recovery. The upper layer is responsible for all of the device-specific commands. The device driver should handle all commands directed towards its specific device by building the necessary sequence of I/O requests to the adapter device driver in order to properly communicate with the device.

These I/O requests contain the commands that are needed by the device. One important aspect to note is that the device driver cannot access any of the adapter resources and should never try to pass the commands directly to the adapter, since it has absolutely no knowledge of the meaning of those commands.

**Related information**

Making an Available Disk a Physical Volume

## FCP, iSCSI, and Virtual SCSI Client Device Drivers

The role of the device driver is to pass information between the operating system and the adapter device driver by accepting I/O requests and passing these requests to the adapter device driver.

The device driver should accept either character or block I/O requests, build the necessary commands, and then issue these commands to the device through the adapter device driver.

The device driver should also process the various required reservations and releases needed for the device. The device driver is notified through the **iodone** kernel service once the adapter has completed the processing of the command. The device driver should then notify its calling process that the request has completed processing through the **iodone** kernel service.

## FCP, iSCSI, and Virtual SCSI Client Adapter Device Driver

Unlike most other device drivers, the adapter device driver does *not* support the **read** and **write** subroutines.

It only supports the **open**, **close**, **ioctl**, **config**, and **strategy** subroutines. Included with the **open** subroutine call is the **openx** subroutine that allows adapter diagnostics.

A device driver does not need to access the diagnostic commands. Commands received from the device driver through the **strategy** routine of the adapter are processed from a queue. Once the command has completed, the device driver is notified through the **iodone** kernel service.

## FCP, iSCSI, and Virtual SCSI Client Adapter and Device Interface

The adapter device driver does not contain the **ddread** and **ddwrite** entry points, but does contain the **ddconfig**, **ddopen**, **ddclose**, **dddump**, and **ddioctl** entry points.

**Note:** Virtual SCSI is available only on IBM eServer™ i5 and IBM eServer™ p5 models.

Therefore, the adapter device driver's entry in the kernel devsw table contains only those entries plus an additional **ddstrategy** entry point. This **ddstrategy** routine is the path that the device driver uses to pass commands to the device driver. Access to these entry points is possible through the following kernel services:

- **fp_open**
- **fp_close**
- **devdump**
- **fp_ioctl**

- **devstrat**

The adapter is accessed by the device driver through the **/dev/fscsi#** special files, where # indicates ascending numbers `0, 1, 2,` and so on. The adapter is designed so that multiple devices on the same adapter can be accessed at the same time.

The iSCSI adapter is accessed by the device driver through the **/dev/iscsi*n*** special files, where *n* indicates ascending numbers `0, 1, 2,` and so on. The adapter is designed so that multiple devices on the same adapter can be accessed at the same time.

The Virtual SCSI Client adapter is accessed by the device driver through the `/dev/vscsi`X special files, where X indicates ascending numbers `0`, `1`, 2, and so on. The adapter is designed such that multiple devices on the same adapter can be accessed at the same time.

For additional information on spanned and gathered write commands, see "Understanding the Execution of SAM Initiator I/O Requests" on page 301.

### *scsi_buf Structure*

The I/O requests made from the device driver to the adapter device driver are completed through the use of the **scsi_buf** structure, which is defined in the **/usr/include/sys/scsi_buf.h** header file.

This structure, which is similar to the **buf** structure in other drivers, is passed between the two subsystem drivers through the **strategy** routine. The following is a brief description of the fields contained in the **scsi_buf** structure:

- Reserved fields should be set to a value of `0`, except where noted.
- The **bufstruct** field contains a copy of the standard **buf** buffer structure that documents the I/O request. Included in this structure, for example, are the buffer address, byte count, and transfer direction. The **b_work** field in the **buf** structure is reserved for use by the adapter device driver. The current definition of the **buf** structure is in the **/usr/include/sys/buf.h** include file.
- The **bp** field points to the original buffer structure received by the Device Driver from the caller, if any. This can be a chain of entries in the case of spanned transfers (commands that transfer data from or to more than one system-memory buffer). A null pointer indicates a nonspanned transfer. The null value specifically tells the adapter device driver that all the information needed to perform the DMA data transfer is contained in the **bufstruct** fields of the **scsi_buf** structure.
- The **scsi_command** field, defined as a **scsi_cmd structure**, contains, for example, the SCSI command length, SCSI command, and a flag variable:
  - The **scsi_length** field is the number of bytes in the actual SCSI command. This is normally `6,10,12,` or 16 (decimal).
  - The **FCP_flags** field contains the following bit flags:

    **SC_NODISC**
    Do not allow the target to disconnect during this command.

    **SC_ASYNC**
    Do not allow the adapter to negotiate for synchronous transfer to the device.

    During normal use, the **SC_NODISC** bit should not be set. Setting this bit allows a device executing commands to monopolize the transport layer. Sometimes it is desirable for a particular device to maintain control of the transport layer once it has successfully arbitrated for it; for instance, when this is the only device on the transport layer or the only device that will be in use. For performance reasons, it might not be desirable to go through selections again to save transport layer overhead on each command.

    Also during normal use, the **SC_ASYNC** bit must not be set. It should be set only in cases where a previous command to the device ended in an unexpected transport free condition. This condition is noted as **SCSI_TRANSPORT_FAULT** in the **adapter_status** field of the **scsi_cmd** structure. Because other errors might also result in the **SCSI_TRANSPORT_FAULT** flag being set, the **SC_ASYNC** bit should only be set on the last retry of the failed command.
  - The `FCP_flags` field is not used by the Virtual SCSI client driver.

- The **scsi_cdb** structure contains the physical SCSI command block. The 6 to 16 bytes of a single SCSI command are stored in consecutive bytes, with the op code identified individually. The **scsi_cdb** structure contains the following fields:
  1. The **scsi_op_code** field specifies the standard op code for this command.
  2. The **scsi_bytes** field contains the remaining command-unique bytes of the command block. The actual number of bytes depends on the value in the **scsi_op_code** field.

- The **timeout_value** field specifies the time-out limit (in seconds) to be used for completion of this command. A time-out value of 0 means no time-out is applied to this I/O request.
- The **status_validity** field contains an output parameter that can have one of the following bit flags as a value:

  **SC_SCSI_ERROR**
  The **scsi_status** field is valid.

  **SC_ADAPTER_ERROR**
  The **adapter_status** field is valid.

- The **scsi_status** field in the **scsi_buf** structure is an output parameter that provides valid command completion status when its **status_validity** bit is nonzero. The **scsi_buf.bufstruct.b_error** field should be set to EIO anytime the **scsi_status** field is valid. Typical status values include:

  **SC_GOOD_STATUS**
  The target successfully completed the command.

  **SC_CHECK_CONDITION**
  The target is reporting an error, exception, or other conditions.

  **SC_BUSY_STATUS**
  The target is currently transporting and cannot accept a command now.

  **SC_RESERVATION_CONFLICT**
  The target is reserved by another initiator and cannot be accessed.

  **SC_COMMAND_TERMINATED**
  The target terminated this command after receiving a terminate I/O process message from the adapter.

  **SC_QUEUE_FULL**
  The target's command queue is full, so this command is returned.

  **SC_ACA_ACTIVE**
  The device has an ACA (auto contingent allegiance) condition that requires a Clear ACA to request to clear it.

- The **adapter_status** field is an output parameter that is valid when its **status_validity** bit is nonzero. The **scsi_buf.bufstruct.b_error** field should be set to EIO anytime the **adapter_status** field is valid. This field contains generic adapter card status. It is intentionally general in coverage so that it can report error status from any typical adapter.

  If an error is detected during execution of a command, and the error prevented the command from actually being sent to the transport layer by the adapter, then the error should be processed or recovered, or both, by the adapter device driver.

  If it is recovered successfully by the adapter device driver, the error is logged, as appropriate, but is not reflected in the **adapter_status** byte. If the error cannot be recovered by the adapter device driver, the appropriate **adapter_status** bit is set and the **scsi_buf** structure is returned to the device driver for further processing.

  If an error is detected after the command was actually sent to the device, then it should be processed or recovered, or both, by the device driver.

  For error logging, the adapter device driver logs transport layer and adapter-related conditions, andl the device driver logs device-related errors. In the following description, a capital letter (A) after the error name indicates that the adapter device driver handles error logging. A capital letter (H) indicates that the device driver handles error logging.

Some of the following error conditions indicate a device failure. Others are transport layer or adapter-related.

**SCSI_HOST_IO_BUS_ERR (A)**
The system I/O transport layer generated or detected an error during a DMA or Programmed I/O (PIO) transfer.

**SCSI_TRANSPORT_FAULT (H)**
The transport protocol or hardware was unsuccessful.

**SCSI_CMD_TIMEOUT (H)**
The command timed out before completion.

**SCSI_NO_DEVICE_RESPONSE (H)**
The target device did not respond to selection phase.

**SCSI_ADAPTER_HDW_FAILURE (A)**
The adapter indicated an onboard hardware failure.

**SCSI_ADAPTER_SFW_FAILURE (A)**
The adapter indicated microcode failure.

**SCSI_FUSE_OR_TERMINAL_PWR (A)**
The adapter indicated a blown terminator fuse or bad termination.

**SCSI_TRANSPORT_RESET (A)**
The adapter indicated the transport layer has been reset.

**SCSI_WW_NAME_CHANGE (A)**
The adapter indicated the device at this SCSI ID has a new world wide name.

**SCSI_TRANSPORT_BUSY (A)**
The adapter indicated the transport layer is busy.

**SCSI_TRANSPORT_DEAD (A)**
The adapter indicated the transport layer currently inoperative and is likely to remain this way for an extended time.

- The **add_status** field contains additional device status. For devices, this field contains the Response code returned.
- When the device driver queues multiple transactions to a device, the **adap_q_status** field indicates whether or not the adapter driver has cleared its queue for this device after an error has occurred. The flag of **SC_DID_NOT CLEAR_Q** indicates that the adapter driver has not cleared its queue for this device and that it is in a halted state (so none of the pending queued transactions are sent to the device).
- The **q_tag_msg** field indicates if the adapter can attempt to queue this transaction to the device. This information causes the adapter to fill in the Queue Tag Message Code of the queue tag message for a command. The following values are valid for this field:

  **SC_NO_Q**
  Specifies that the adapter does not send a queue tag message for this command, and so the device does not allow more than one command on its command queue. This value must be used for all commands sent to devices that do not support command tag queuing.

  **SC_SIMPLE_Q**
  Specifies placing this command in the device's command queue. The device determines the order that it executes commands in its queue. The SCSI-2 specification calls this value the `Simple Queue Tag Message`.

  **SC_HEAD_OF_Q**
  Specifies placing this command first in the device's command queue. This command does not preempt an active command at the device, but it is executed before all other commands in the command queue. The SCSI-2 specification calls this value the `Head of Queue Tag Message`.

  **SC_ORDERED_Q**
  Specifies placing this command in the device's command queue. The device processes these commands in the order that they are received. The SCSI-2 specification calls this value the `Ordered Queue Tag Message`.

**SC_ACA_Q**
Specifies placing this command in the device's command queue, when the device has an ACA (auto contingent allegiance) condition. The SCSI-3 Architecture Model calls this value the `ACA task attribute`.

**Note:** Commands with the value of `SC_NO_Q` for the **q_tag_msg** field (except for request sense commands) should not be queued to a device whose queue contains a command with another value for **q_tag_msg**. If commands with the `SC_NO_Q` value (except for request sense) are sent to the device, then the device driver must make sure that no active commands are using different values for **q_tag_ms**. Similarly, the device driver must also make sure that a command with a **q_tag_msg** value of `SC_ORDERED_Q`, `SC_HEAD_Q`, or `SC_SIMPLE_Q` is not sent to a device that has a command with the **q_tag_msg** field of `SC_NO_Q`.

- The flags field contains bit flags sent from the device driver to the adapter device driver. The following flags are defined:

**SC_RESUME**
When set, means the adapter device driver should resume transaction queuing for this ID/LUN. Error recovery is complete after a **SCIOLHALT** operation, check condition, or severe transport error. This flag is used to restart the adapter device driver following a reported error.

**SC_DELAY_CMD**
When set, means the adapter device driver should delay sending this command (following a reset or BDR to this device) by at least the number of seconds specified to the adapter device driver in its configuration information. For devices that do not require this function, this flag should not be set.

**SC_Q_CLR**
When set, means the adapter driver should clear its transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual command in the **scsi_buf** because it is flushed back to the device driver with the rest of the transactions for this ID/LUN. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and LUN. This flag is valid only during error recovery of a check condition or command terminated at a command tag queuing device when the **SC_DID_NOT_CLR_Q** flag is set in the **scsi_buf.adap_q_status** field.

**SC_Q_RESUME**
When set, means that the adapter driver should resume its halted transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual command to be sent to the adapter driver. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). If the transaction containing this flag setting is the first issued by the device driver after it receives an error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set also.

**SC_CLEAR_ACA**
When set, means the SCSI adapter driver should issue a Clear ACA task management request for this ID/LUN. This flag should be used in conjunction with either the **SC_Q_CLR** or **SC_Q_RESUME** flags to clear or resume the SCSI adapter driver's queue for this device. If neither of these flags is used, then this transaction is treated as if the **SC_Q_RESUME** flag is also set. The transaction containing the **SC_CLEAR_ACA** flag setting does not require an actual SCSI command in the **sc_buf**. If this transaction contains a SCSI command then it will be processed depending on whether **SC_Q_CLR** or **SC_Q_RESUME** is set. This transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and LUN. This flag is valid only during error recovery of a check condition or command terminated at a command tag queuing.

**SC_TARGET_RESET**
When set, means the SCSI adapter driver should issue a Target Reset task management request for this ID/LUN. This flag should be used in conjunction with ethe **SC_Q_CLR** flag flag.The transaction containing this flag setting does allow an actual command to be sent to the adapter driver. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) filled in with the device's SCSI ID. If the transaction containing this flag setting is the first issued by the device driver after it receives an error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set also.

**SC_LUN_RESET**
When set, means the SCSI adapter driver should issue a Lun Reset task management request for this ID/LUN. This flag should be used in conjunction with ethe **SC_Q_CLR** flag flag.The transaction containing this flag setting does allow an actual command to be sent to the adapter driver. However, this transaction must have the the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). If the transaction containing this flag setting is the first issued by the device driver after it receives an error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set also.

- The **dev_flags** field contains additional values sent from the FCP device driver to the FCP adapter device driver. This field is not used for iSCSI or Virtual SCSI device drivers. The following values are defined:

  **FC_CLASS1**
  When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 1 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

  **FC_CLASS2**
  When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 2 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

  **FC_CLASS3**
  When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 3 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

  **FC_CLASS4**
  When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 4 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

- The **add_work** field is reserved for use by the adapter device driver.
- The **adap_set_flags** field contains an output parameter that can have one of the following bit flags as a value:

  **SC_AUTOSENSE_DATA_VALID**
  Autosense data was placed in the autosense buffer referenced by the **autosense_buffer_ptr** field.

- The **autosense_length** field contains the length in bytes of the SCSI device driver's sense buffer, which is referenced via the **autosense_buffer_ptr** field. For devices this field must be non-zero, otherwise the autosense data will be lost.
- The **autosense_buffer_ptr** field contains the address of the SCSI devices driver's autosense buffer for this command. For devices this field must be non-NULL, otherwise the autosense data will be lost.
- The **dev_burst_len** field contains the burst size if this write operation in bytes. This should only be set by the device driver if it has negotiated with the device and it allows burst of write data without transfer readys. For most operations, this should be set to 0.
- The **scsi_id** field contains the 64-bit SCSI ID for this device. This field must be set for FCP devices.
- The **lun_id** field contains the 64-bit lun ID for this device. This field must be set for devices.
- The **kernext_handle** field contains the pointer returned from the **kernext_handle** field of the **scsi_sciolst** argument for the SCIOLSTART ioctl.

### *Adapter and Device Driver Intercommunication*

This section describes the communication between the adapter and the device drivers through the various routines.

In a typical request to the device driver, a call is first made to the device driver's **strategy** routine, which takes care of any necessary queuing. The device driver's **strategy** routine then calls the device driver's **start** routine, which fills in the **scsi_buf** structure and calls the adapter driver's **strategy** routine through the **devstrat** kernel service.

The adapter driver's **strategy** routine validates all of the information contained in the **scsi_buf** structure and also performs any necessary queuing of the transaction request. If no queuing is necessary, the adapter driver's **start** subroutine is called.

When an interrupt occurs, adapter driver **interrupt** routine fills in the **status_validity** field and the appropriate **scsi_status** or **adapter_status** field of the **scsi_buf** structure. The **bufstruct.b_resid** field is also filled in with the value of nontransferred bytes. The adapter driver's **interrupt** routine then passes this newly filled in **scsi_buf** structure to the **iodone** kernel service, which then signals the device driver's **iodone** subroutine. The adapter driver's **start** routine is also called from the **interrupt** routine to process any additional transactions on the queue.

The device driver's **iodone** routine should then process all of the applicable fields in the queued **scsi_buf** structure for any errors and attempt error recovery if necessary. The device driver should then dequeue the **scsi_buf** structure and then pass a pointer to the structure back to the **iodone** kernel service so that it can notify the originator of the request.

## FCP, iSCSI, and Virtual SCSI Client Adapter Device Driver Routines

This overview lists the FCP, iSCSI, and virtual SCSI client adapter device driver routines.

### *config Routine*

The **config** routine performs all of the processing needed to configure, unconfigure, and read Vital Product Data (VPD) for the adapter.

When this routine is called to configure an adapter, it performs the required checks and building of data structures needed to prepare the adapter for the processing of requests.

When asked to unconfigure or terminate an adapter, this routine deallocates any structures defined for the adapter and marks the adapter as unconfigured. This routine can also be called to return the Vital Product Data for the adapter, which contains information that is used to identify the serial number, change level, or part number of the adapter.

### *open Routine*

The **open** routine establishes a connection between a special file and a file descriptor.

This file descriptor is the link to the special file that is the access point to a device and is used by all subsequent calls to perform I/O requests to the device. Interrupts are enabled and any data structures needed by the adapter driver are also initialized.

### *close Routine*

This section describes about the **close** routine.

The **close** routine marks the adapter as closed and disables all future interrupts, which causes the driver to reject all future requests to this adapter.

### *openx Routine*

The **openx** routine allows a process with the proper authority to open the adapter in diagnostic mode.

If the adapter is already open in either normal or diagnostic mode, the **openx** subroutine has a return value of `-1`. Improper authority results in an **errno** value of EPERM, while an already open error results in an **errno** value of EACCES. If the adapter is in diagnostic mode, only the **close** and **ioctl** routines are allowed. All other routines return a value of `-1` and an **errno** value of EACCES.

While in diagnostics mode, the adapter can run diagnostics, run wrap tests, and download microcode. The **openx** routine is called with an *ext* parameter that contains the adapter mode and the SC_DIAGNOSTIC value, both of which are defined in the **sys/scsi.h** header file.

### *strategy Routine*

The **strategy** routine is the link between the device driver and the adapter device driver for all normal I/O requests.

Whenever the device driver receives a call, it builds an **scsi_buf** structure with the correct parameters and then passes it to this routine, which in turn queues up the request if necessary. Each request on the pending queue is then processed by building the necessary commands required to carry out the request. When the command has completed, the device driver is notified through the **iodone** kernel service.

### *ioctl Routine*

The **ioctl** routine allows various diagnostic and nondiagnostic adapter operations.

Operations include the following:

- **IOCINFO**
- **SCIOLSTART**
- **SCIOLSTOP**
- **SCIOLINQU**
- **SCIOLEVENT**
- **SCIOLSTUNIT**
- **SCIOLTUR**
- **SCIOLREAD**
- **SCIOLRESET**
- **SCIOLHALT**
- **SCIOLCMD**
- **SCIOLCHBA**
- **SCIOLPASSTHRUHBA**

### *start Routine*

The **start** routine is responsible for checking all pending queues and issuing commands to the adapter. When a command is issued to the adapter, the **scsi_buf** is converted into an adapter specific request needed for the **scsi_buf**.

At this time, the **bufstruct.b_addr** for the **scsi_buf** will be mapped for DMA. When the adapter specific request is completed, the adapter will be notified of this request.

### *interrupt Routine*

The **interrupt** routine is called whenever the adapter posts an interrupt.

When this occurs, the interrupt routine will find the **scsi_buf** corresponding to this interrupt. The buffer for the **scsi_buf** will be unmapped from DMA. If an error occurred, the **status_validity**, **scsi_status**, and **adapter_status** fields will be set accordingly. The **bufstruct.b_resid** field will be set with the number of nontransferred bytes. The interrupt handler then runs the **iodone** kernel service against the **scsi_buf**, which will send the **scsi_buf** back to the device driver which originated it.

## SAM Adapter ioctl Operations

This overview lists the SAM adapter ioctl operations.

**Related concepts**

SAM Adapter Device Driver ioctl Commands

Various ioctl operations must be performed for proper operation of the adapter device driver.

### *IOCINFO for FCP Adapters*

This operation lets an FCP device driver obtain important information about a FCP adapter, including the adapter's SCSI ID, the maximum data transfer size in bytes, and the FC topology to which the adapter is connected.

By knowing the maximum data transfer size, a FCP device driver can control several different devices on several different adapters. This operation returns a **devinfo** structure as defined in the **sys/devinfo.h** header file with the device type **DD_BUS** and subtype **DS_FCP**. The following is an example of a call to obtain the information:

```
rc = fp_ioctl(fp, IOCINFO, &infostruct, NULL);
```

where *fp* is a pointer to a file structure and *infostruct* is a **devinfo** structure. A non-zero `rc` value indicates an error. Note that the **devinfo** structure is a union of several structures and that **fcp** is the structure that applies to the adapter. For example, the maximum transfer size value is contained in the **infostruct.un.fcp.max_transfer** variable and the card ID is contained in **infostruct.un.fcp.scsi_id**.

### *IOCINFO for iSCSI Adapters*

This operation lets an iSCSI device driver obtain important information about an iSCSI adapter, including the adapter's maximum data transfer size in bytes.

By knowing the maximum data transfer size, an iSCSI device driver can control several different devices on several different adapters. This operation returns a **devinfo** structure as defined in the **sys/devinfo.h** header file with the device type **DD_BUS** and subtype **DS_ISCSI**. The following is an example of a call to obtain the information:

```
rc = fp_ioctl(fp, IOCINFO, &infostruct, NULL);
```

where *fp* is a pointer to a file structure and *infostruct* is a **devinfo** structure. A non-zero rc value indicates an error. Note that the **devinfo** structure is a union of several structures and that **iscsi** is the structure that applies to the adapter. For example, the maximum transfer size value is contained in the *infostruct.un.iscsi.max_transfer* variable.

### *IOCINFO for SAS Adapters*

This operation allows a SAS device driver to obtain important information about a SAS adapter, such as the maximum data-transfer size in bytes of the adapter.

By knowing the maximum data transfer size, a SAS device driver can control several different devices on several different adapters. This operation returns a **devinfo** structure as defined in the **sys/devinfo.h** header file with the **DD_BUS** device type and the **DS_SAS** subtype. The following example shows a call to obtain the information:

```
rc = fp_ioctl(fp, IOCINFO, &infostruct, NULL);
```

where the *fp* parameter points to a file structure and the *infostruct* parameter is a **devinfo** structure. A nonzero rc value indicates an error. The **devinfo** structure is a union of several structures and the **sas** structure applies to the adapter. For example, the maximum transfer-size value is contained in the *infostruct.un.sas.max_transfer* variable.

### *IOCINFO for Virtual SCSI Adapters*

This operation lets a Virtual SCSI device driver obtain important information about a Virtual SCSI adapter, including the adapter's maximum data transfer size in bytes.

This information is determined by the Virtual SCSI server driver. By knowing the maximum data transfer size, a Virtual SCSI device driver can control several different devices on several different adapter instances. This operation returns a `devinfo` structure as defined in the `sys/devinfo.h` header file

with the device type DD_BUS and subtype DS_CVSCSI. The following is an example of a call to obtain the information:

```
rc = fp_ioctl(fp, IOCINFO, &infostruct, NULL);
```

where fp is a pointer to a file structure and infostruct is a devinfo structure. A non-zero rc value indicates an error. The devinfo structure is a union of several structures and Virtual SCSI is the structure that applies to the adapter. For example, the maximum transfer size value is contained in the *infostruct.un.vscsi.max_transfer* variable.

### *SCIOLSTART*

This operation opens a logical path to the target device and causes the SAM adapter device driver to allocate and initialize all of the data areas needed for the target device. The **SCIOLSTOP** operation should be issued when those data areas are no longer needed. The **SCIOLSTART** operation must be issued before any nondiagnostic operation except for **IOCINFO**.

TThe following is a typical call:

```
rc = fp_ioctl(fp, SCIOLSTART, &sciolst);
```

where *fp* is a pointer to a file structure and *sciolst* is a **scsi_sciolst** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI and Logical Unit Number (LUN) ID values of the device to be started. In addition, the **scsi_sciolst** structure can be used to specify an explicit login for this operation.

For FCP adapters, the **version** field of the **scsi_sciolst** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If the **world_wide_name** field is set and the **version** field is set to SCSI_VERSION_1, the World Wide Name can be used to address the target instead of the **scsi_id** field. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. For AIX® 5.2 through AIX® 5.2.0.9, if the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**.

  If a World Wide Name or Node Name is provided and it does not match the World Wide Name or Node Name that was detected for the target, an error log is generated and the **SCIOLSTART** operation fails with an errno of **ENXIO**.

  Upon successfully return from an **SCIOLSTART** operation, both the **world_wide_name** field and the **node_name** field are set to the World Wide Name and Node Name of this device. These values are inspected to ensure that the **SCIOLSTART** operation was delivered to the intended device.

  If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **parms.fcp_target** - If the caller is using SCSI_VERSION_1 , it can set the FCP_TARGET_VALID flag to indicate that the **parms.fcp_target** union is used.
- **parms.fcp_target.flags** - The caller can set the SCSI_CONC_DYNTRK flag to indicate that it supports concurrent update of the dynamic tracking state. If the Fibre Channel driver also supports this then it responds by setting the SCSI_CONC_DYNTRK_CONF flag. If both parties support, any future **scsi_bufs** for this device which complete with **adapter_status** of SCSI_WW_NAME_CHANGE will carry an **add_adap_status** value of 0 or SCSI_ADAP_CONC_DYNTRK_ON to indicate whether dynamic tracking is currently disabled or enabled.

For iSCSI adapters, this version field of the **scsi_sciolst** must be set to a minimum value of **SCSI_VERSION_1** (defined in the **/usr/include/sys/scsi_buf.h** file). In addition, iSCSI adapters require the caller to set the following fields:

- **lun_id** of the device's LUN ID
- **parms.iscsi.name** to the device's iSCSI target name
- **parms.iscsi.iscsi_ip_addr** to the device's IP V4 or IP V6 address
- **parms.iscsi.port_num** to the devices TCP port number

If the iSCSI **SCIOLSTART** ioctl operation completes successfully, then the **adap_set_flags** field should have the **SCIOL_RET_ID_ALIAS** flag and the **scsi_id** field set to a SCSI ID alias that can be used for subsequent ioctl calls to this device other than **SCIOLSTART**.

For Virtual SCSI adapters, the version field of the `scsi_sciolst` structure must be set to the value of `SCSI_VERSION_1` (defined in the `/usr/include/sys/scsi_buf.h` file). In addition, Virtual SCSI adapters require the caller to set the `lun_id` field to the Logical Unit Id (LUN) of the device being started.

For SAS adapters, the **version** field of the **scsi_sciolst** structure must be set to a minimum value of SCSI_VERSION_1 (defined in the /usr/include/sys/scsi_buf.h file). In addition, SAS adapter device drivers require the caller to set the following fields:

- **scsi_id** (The field can also be referred to as the SAS address.)
- **lun_id**

For AIX 5.2, if the FCP **SCIOLSTART** ioctl operation completes successfully, and the **adap_set_flags** field has the **SCIOL_DYNTRK_ENABLED** flag set, **Dynamic Tracking of FC Devices** has been enabled for this device.

All FC adapter ioctl calls for AIX 5.2, must set the **version** field to **SCSI_VERSION_1** if indicated in the ioctl structure comments in the header files. The **world_wide_name** and **node_name** fields of all **SCSI_VERSION_1** ioctl structures must also be set. This is important if dynamic tracking has been enabled on this adapter. By using dynamic tracking, the FC adapter driver can recover from **scsi_id** changes of FC devices while devices are online. Because the **scsi_id** can change, use the **world_wide_name** and **node_name** fields to ensure communication with the intended device.

Failure to use a **SCSI_VERSION_1** ioctl structure for **SCIOLSTART** when dynamic tracking is enabled can produce undesired results, and temporarily disable dynamic tracking for a given device. If a target has at least one lun activated by **SCIOLSTART** with the version field set to **SCSI_VERSION_1**, then a **SCSI_VERISON_0 SCIOLSTART** fails. If this is the first lun activated by **SCIOLSTART** on this target and the version field is set to **SCSI_VERSION_0**, then an error log of type **INFO** is generated and dynamic tracking is temporarily disabled for this target until a corresponding **SCSI_VERSION_0 SCIOLSTOP** is issued.

The **version** field for all ioctl structures should be set consistently. For example, if an **SCIOLSTART** operation is performed with the version field set to **SCSI_VERSION_1**, but the **SCIOLINQU** or **SCIOLSTOP** ioctl operations have the **version** field set to **SCSI_VERSION_0**, then the ioctl call fails if dynamic tracking is enabled because the version fields do not match.

If the FCP **SCIOLSTART** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SCIOL_RET_ID_ALIAS** flag set. This field is set only if the **world_wide_name** field was provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field contains the new **scsi_id** value.

If the caller of the SAM **SCIOLSTART** is a kernel extension, the **SCIOL_RET_HANDLE** flag can be set in the **adap_set_flags** field along with the **kernext_handle** field. In this case the **kernext_handle** field can be used for **scsi_buf** structures issued to the adapter driver for this device.

A nonzero return value indicates an error has occurred and all operations to this SCSI/LUN pair should cease because the device is either already started or failed the start operation. Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | The command could not complete due to a system error. |

| Item | Description |
|---|---|
| EINVAL | Either the Logical Unit Number (LUN) ID or SCSI ID is invalid, or the adapter is already open. |
| ENOMEM | Indicates that system resources are not available to start this device. |
| ETIMEDOUT | Indicates that the command did not complete. |
| ENODEV | Indicates that no device responded to the explicit process login at this SCSI ID. |
| ECONNREFUSED | Indicates that the device at this SCSI ID rejected explicit process login. This could be due to the device rejecting the security password or the device does not support FCP. |
| EACCES | The adapter is not in normal mode. |

### *SCIOLSTOP*

This operation closes a logical path to the device and causes the adapter device driver to deallocate all data areas that were allocated by the **SCIOLSTART** operation. This operation should only be issued after a successful **SCIOLSTART** operation to a device.

The following is a typical call:

```
rc = fp_ioctl(fp, SCIOLSTOP, &sciolst);
```

where *fp* is a pointer to a file structure and *sciolst* is a **scsi_sciolst** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI or iSCSI device's SCSI ID alias, and Logical Unit Number (LUN) ID values of the device to be stopped.

A nonzero return value indicates an error has occurred. Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | An unrecoverable system error has occurred. |
| EINVAL | The adapter was not in open mode. |

The **version** field of the **scsi_sciolst** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. For AIX® 5.2 through AIX® 5.2.0.9, if the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**. If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.

For Virtual SCSIadapters, the version field of the `scsi_sciolst` structure must be set to the value of SCSI_VERSION_1 (defined in the `/usr/include/sys/scsi_buf.h` file). In addition, Virtual SCSI adapters require the caller to set the `lun_id` field to the Logical Unit Id (LUN) of the device being stopped.

This operation requires **SCIOLSTART** to be run first.

### *SCIOLEVENT*

This operation lets a device driver register a particular device instance for receiving asynchronous event status by calling the **SCIOLEVENT** ioctl operation for the adapter device driver. When an event covered by the **SCIOLEVENT** ioctl operation is detected by the adapter device driver, it builds an **scsi_event_info**

structure and passes a pointer to the structure and to the asynchronous event-handler routine entry point, which was previously registered.

The information reported in the **scsi_event_info.events** field does not queue to the device driver, but is instead reported as one or more flags as they occur. Because the data does not queue, the adapter device driver writer can use a single **scsi_event_info** structure and pass it one at a time, by pointer, to each asynchronous event handler routine for the appropriate device instance. After determining for which device the events are being reported, the device driver must copy the **scsi_event_info.events** field into local space and must not modify the contents of the rest of the **scsi_event_info** structure.

Because the event status is optional, the device driver writer determines what action is necessary to take upon receiving event status. The writer might decide to save the status and report it back to the calling application, or the device driver or application level program can take error recovery actions.

This operation should only be issued after a successful **SCIOLSTART** operation to a device. The following is a typical call:

```
rc = fp_ioctl(fp, SCIOLEVENT, &scevent);
```

where *fp* is a pointer to a file structure and *scevent* is a **scsi_event_struct** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI and Logical Unit Number (LUN) ID values of the device to be started.

A non-zero return value indicates an error has occurred. Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | An unrecoverable system error has occurred. |
| EINVAL | The adapter was not in open mode. |

For FCP adapters, the **version** field of the **scsi_event_struct** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. If the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**. If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.

This operation requires **SCIOLSTART** to be run first.

If the FCP **SCIOLEVENT** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SC_RET_ID** flag set. This field is set only if the **world_wide_name** and **node_ name** fields were provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field contains the new **scsi_id** value.

### *SCIOLINQU*

This operation issues an inquiry command to an device and is used to aid in device configuration.

The following is a typical call:

```
rc = ioctl(adapter, SCIOLINQU, &inquiry_block);
```

where *adapter* is a file descriptor and *inquiry_block* is a **scsi_inquiry** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The FCP SCSI ID or device's SCSI ID alias, and LUN must be placed in the **scsi_inquiry** parameter block. Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | A system error has occurred. Consider retrying the operation several times, because another attempt might be successful. |
| EFAULT | A user process copy has failed. |
| EINVAL | The device is not opened. |
| EACCES | The adapter is in diagnostics mode. |
| ENOMEM | A memory request has failed. |
| ETIMEDOUT | The command has timed out. Consider retrying the operation several times, because another attempt might be successful. |
| ENODEV | The device is not responding. Possibly no LUNs exist on the present FCP ID. |
| ENOCONNECT | A transport fault occurred. |

For all physical transport types, the **version** field of the **scsi_inquiry** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. If the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**. If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.

When the SCIOLINQU ioctl request with the `version` field set to SCSI_VERSION_2 completes and the device did not fully satisfy the request, the `residual` field indicates left over data. If the request completes successfully, the `residual` field indicates the device does not have all the requested data. If the request did not complete successfully, check the `status_validity` to see whether a valid SCSI problem exists. If a valid SCSI problem exists, the `residual` field indicates the number of bytes by which the device failed to complete the request.

This operation requires **SCIOLSTART** to be run first.

If the FCP **SCIOLINQU** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SC_RET_ID** flag set. This field is set only if the **world_wide_name** and **node_ name** fields were provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field contains the new **scsi_id** value.

### *SCIOLSTUNIT*

This operation issues a start unit command to an device and is used to aid in device configuration.

The following is a typical call:

```
rc = ioctl(adapter, SCIOLSTUNIT, &start_block);
```

where *adapter* is a file descriptor and *start_block* is a **scsi_startunit** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The FCP ID or iSCSI device's SCSI ID alias, and LUN should be placed in the **scsi_startunit** parameter block. The **start_flag** field designates the start option, which when set to `true`, makes the device available for use. When this field is set to `false`, the device is stopped.

The **immed_flag** field supports overlapping start operations to several devices on the adapter. When this field is set to `false`, status is returned only when the operation has completed. When this field is set to

`true`, status is returned as soon as the device receives the command. The **SCIOLTUR** operation can then be issued to check on completion of the operation on a particular device.

**Attention:** When the SAM adapter issues simultaneous start operations, it is important that a sufficient delay is buffered between successive **SCIOLSTUNIT** operations to devices sharing a common power supply because damage to the system or devices can occur.

Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | A system error has occurred. Consider retrying the operation several times, because another attempt might be successful. |
| EFAULT | A user process copy has failed. |
| EINVAL | The device is not opened. |
| EACCES | The adapter is in diagnostics mode. |
| ENOMEM | A memory request has failed. |
| ETIMEDOUT | The command has timed out. Consider retrying the operation several times, because another attempt might be successful. |
| ENODEV | The device is not responding. Possibly no LUNs exist on the present FCP ID. |
| ENOCONNECT | A transport fault occurred. |

For all physical transport types, the **version** field of the **scsi_startunit** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. If the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**. If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.

This operation requires **SCIOLSTART** to be run first.

If the FCP **SCIOLSTUNIT** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SC_RET_ID** flag set. This field is set only if the **world_wide_name** and **node_ name** fields were provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field contains the new **scsi_id** value.

### *SCIOLTUR*

This operation issues a Test Unit Ready command to an adapter and aids in device configuration.

The following is a typical call:

```
rc = ioctl(adapter, SCIOLTUR, &ready_struct);
```

where *adapter* is a file descriptor and *ready_struct* is a **scsi_ready** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The FCP SCSI ID or iSCSI device's SCSI ID alias, and LUN should be placed in the **scsi_ready** parameter block. The status of the device can be determined by evaluating the two output fields: **status_validity** and **scsi_status**. Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | A system error has occurred. Consider retrying the operation several (around three) times, because another attempt might be successful. If an EIO error occurs and the **status_validity** field is set to SC_FCP_ERROR, then the **scsi_status** field has a valid value and should be inspected.<br><br>If the **status_validity** field is zero and remains so on successive retries, then an unrecoverable error has occurred with the device.<br><br>If the **status_validity** field is SC_FCP_ERROR and the **scsi_status** field contains a Check Condition status, then the **SCIOLTUR** operation should be retried after several seconds.<br><br>If after successive retries, the Check Condition status remains, the device should be considered inoperable. |
| EFAULT | A user process copy has failed. |
| EINVAL | The device is not opened. |
| EACCES | The adapter is in diagnostics mode. |
| ENOMEM | A memory request has failed. |
| ETIMEDOUT | The command has timed out. Consider retrying the operation several times, because another attempt might be successful. |
| ENODEV | The device is not responding and possibly no LUNs exist on the present target. |
| ENOCONNECT | A transport fault occurred. |

For FCP adapters, the **version** field of the **scsi_ready** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. If the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**. If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.

For Virtual SCSI adapters, the version field of the scsi_sciolst structure must be set to the value of SCSI_VERSION_1 (defined in the /usr/include/sys/scsi_buf.h file). In addition, Virtual SCSI adapters require the caller to set the lun_id field to the Logical Unit Id (LUN) of the device being reset. Target Reset is not supported on vscsi devices, so the driver sends a Lun Reset regardless of the value of the SCIOLRESET_LUN_RESET flag.

This operation requires **SCIOLSTART** to be run first.

If the FCP **SCIOLTUR** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SC_RET_ID** flag set. This field is set only if the **world_wide_name** and **node_ name** fields were provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field contains the new **scsi_id** value.

### *SCIOLREAD*

This operation issues an read command to an device and is used to aid in device configuration.

The following is a typical call:

```
rc = ioctl(adapter, SCIOLREAD, &readblk);
```

where *adapter* is a file descriptor and *readblk* is a **scsi_readblk** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The FCP SCSI ID or iSCSI device's SCSI ID alias, and the LUN should be placed in the **scsi_readblk** parameter block. Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | A system error has occurred. Consider retrying the operation several times, because another attempt might be successful. |
| EFAULT | A user process copy has failed. |
| EINVAL | The device is not opened. |
| EACCES | The adapter is in diagnostics mode. |
| ENOMEM | A memory request has failed. |
| ETIMEDOUT | The command has timed out. Consider retrying the operation several times, because another attempt might be successful. |
| ENODEV | The device is not responding. Possibly no LUNs exist on the present target. |
| ENOCONNECT | A transport fault occurred. |

For all transport types, the **version** field of the **scsi_readblk** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. If the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**. If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.

When the SCIOLREAD ioctl request with the `version` field set to SCSI_VERSION_2 completes and the device did not fully satisfy the request, the `residual` field indicates left over data. If the request completes successfully, the `residual` field indicates the device does not have all the requested data. If the request did not complete successfully, check the `status_validity` to see whether a valid SCSI bus problem exists. If a valid SCSI bus problem exists, the `residual` field indicates the number of bytes by which the device failed to complete the request.

This operation requires **SCIOLSTART** to be run first.

If the FCP **SCIOLREAD** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SC_RET_ID** flag set. This field is set only if the **world_wide_name** and **node_ name** fields were provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field contains the new **scsi_id** value.

### *SCIOLRESET*

If the **SCIOLRESET_LUN_RESET** flag is not set in the flags field of the **scsi_sciolst**, then this operation causes an FCP or iSCSI device, clear all current commands, and return to an initial state by issuing a Target Reset, which resets all LUNs associated with the specified FCP ID or iSCSI device's SCSI ID alias. If, for an FCP or iSCSI device the **SCIOLRESET_LUN_RESET** flag is set in the flags field of the **scsi_sciolst**, then this operation causes an FCP or iSCSI device to clear all current commands, and return to an initial state by issuing a Lun Reset, which resets just the specified LUN associated with the specified FCP ID or iSCSI device's SCSI ID alias. For the SAS transport type, the Target Reset command

is not defined; and thus for SAS, this ioctl always results in Lun Reset being issued without regard to the **SCIOLRESET_LUN_RESET** flag setting.

**Note:** Both the Target and LUN reset commands that are generated by the SCIOLRESET ioctl causes a device that honors a standard SCSI reservation to release that reservation. Whether a device driver for the device reestablishes the standard SCSI reservation is based on the following conditions:

- The configuration attributes of the device
- The condition when the reservation is initially established
- The mode in which the device is opened

The following is a typical call:

```
        rc = fp_ioctl(fp, SCIOLRESET, &sciolst);
```

where *fp* is a pointer to a file structure and *sciolst* is a **scsi_sciolst** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI ID or iSCSI device's SCSI ID alias, and Logical Unit Number (LUN) ID values of the device to be started.

A nonzero return value indicates an error has occurred. Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | An unrecoverable system error has occurred. |
| EINVAL | The device is not opened. |
| EACCES | The adapter is in diagnostics mode. |
| ETIMEDOUT | The operation did not complete before the time-out value was exceeded. |

For all transport types, the **version** field of the **scsi_sciolst** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. If the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**. If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.

This operation requires **SCIOLSTART** to be run first.

If the FCP **SCIOLRESET** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SCIOL_RET_ID_ALIAS** flag set. This field is set only if the **world_wide_name** and **node_ name** fields were provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field contains the new **scsi_id** value.

### *SCIOLHALT*

This operation stops the current command of the selected device, clears the command queue of any pending commands, and brings the device to a halted state. The adapter sends a SCSI abort message to the device and is usually used by the device driver to abort the current operation instead of waiting for it to complete or time out.

The adapter also performs normal error recovery procedures during this command. The following is a typical call:

```
rc = fp_ioctl(fp, SCIOLHALT, &sciolst);
```

where *fp* is a pointer to a file structure and *sciolst* is a **scsi_sciolst** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI ID or iSCSI device's SCSI ID alias, and Logical Unit Number (LUN) ID values of the device to be started.

A nonzero return value indicates an error has occurred. Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | An unrecoverable system error has occurred. |
| EINVAL | The device is not opened. |
| EACCES | The adapter is in diagnostics mode. |
| ETIMEDOUT | The operation did not complete before the time-out value was exceeded. |

For all transport types, the **version** field of the **scsi_sciolst** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. If the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**. If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.

This operation requires **SCIOLSTART** to be run first.

If the FCP **SCIOLHALT** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SCIOL_RET_ID_ALIAS** flag set. This field is set only if the **world_wide_name** and **node_ name** fields were provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field contains the new **scsi_id** value.

### *SCIOLCMD*

After the SCSI device has been successfully started using **SCIOLSTART**, the **SCIOLCMD** ioctl operation provides the means for issuing any SCSI command to the specified device. The SAM adapter driver performs no error recovery or logging on failures of this ioctl operation.

The following is a typical call:

```
rc = ioctl(adapter, SCIOLCMD, &iocmd);
```

where *adapter* is a file descriptor and *iocmd* is a **scsi_iocmd** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The SCSI ID or iSCSI device's SCSI ID alias, and LUN ID should be placed in the **scsi_iocmd** parameter block.

The SCSI status byte and the adapter status bytes are returned using the **scsi_iocmd** structure. If the **SCIOLCMD** operation returns a value of -1 and the **errno** global variable is set to a nonzero value, the requested operation has failed. In this case, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum, a value of -1 is returned and the **errno** global variable set to a value of **EINVAL.** Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device to get request sense information.

Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | A system error has occurred. Consider retrying the operation several (around three) times, because another attempt might be successful. If an EIO error occurs and the **status_validity** field is set to SC_SCSI_ERROR, then the **scsi_status** field has a valid value and should be inspected.<br><br>If the **status_validity** field is zero and remains so on successive retries then an unrecoverable error has occurred with the device.<br><br>If the **status_validity** field is SC_SCSI_ERROR and the **scsi_status** field contains a Check Condition status, then a SCSI request sense should be issued using the **SCIOLCMD** ioctl to recover the the sense data. |
| EFAULT | A user process copy has failed. |
| EINVAL | The device is not opened. |
| EACCES | The adapter is in diagnostics mode. |
| ENOMEM | A memory request has failed. |
| ETIMEDOUT | The command has timed out. Consider retrying the operation several times, because another attempt might be successful. |
| ENODEV | The device is not responding. |
| ETIMEDOUT | The operation did not complete before the time-out value was exceeded. |

For all transport types, the **version** field of the **scsi_iocmd** structure must be set to a minimum value of SCSI_VERSION_1, which is defined in the **/usr/include/sys/scsi_buf.h** file. In addition, the following fields can be set:

- **world_wide_name** - The caller can set the **world_wide_name** field to the World Wide Name of the attached target device. If **Dynamic Tracking of FC devices** is enabled, the **world_wide_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.
- **node_name** - The caller can set the **node_name** field to the Node Name of the attached target device. If the **world_wide_name** field and the **version** field are set to SCSI_VERSION_1 but the **node_name** field is not set, the **scsi_id** field is used for device lookup instead of the **world_wide_name**. If **Dynamic Tracking of FC devices** is enabled, the **node_name** field must be set to ensure communication with the device because the **scsi_id** field of a device can change after dynamic tracking events.

The version field of the scsi_iocmd structure can be set to the value of SCSI_VERSION_2, and the user can provide the following fields:

- variable_cdb_ptr - pointer to a buffer that contains the SCSI variable*cdb*.
- variable_cdb_length - the length of the *cdb* variable to which the *variable_cdb_ptr* points.

**Note:** The SAS transport type does not support the *cdbs* variable length.

When the SCIOLCMD ioctl request with the version field set to SCSI_VERSION_2 completes and the device did not fully satisfy the request, the residual field indicates left over data. If the request completes successfully, the residual field indicates the device does not have all the requested data. If the request did not complete successfully, check the status_validity to see whether a valid SCSI problem exists. If a valid SCSI problem exists, the residual field indicates the number of bytes by which the device failed to complete the request.

This operation requires **SCIOLSTART** to be run first.

If the FCP **SCIOLCMD** ioctl operation completes successfully, then the **adap_set_flags** field might have the **SC_RET_ID** flag set. This field is set only if the **world_wide_name** and **node_ name** fields were

provided in the ioctl call and the FC adapter driver detects that the **scsi_id** field of this device has changed. The **scsi_id** field contains the new **scsi_id** value.

### *SCIOLNMSRV*

This operation returns a list of target devices and is used to aid in SCSI device configuration.

**Note: SCIOLNMSRV** is specific to the iSCSI and FCP transport types.

The following is a typical call:

```
rc = ioctl(adapter, SCIOLNMSRV, &nmserv);
```

where *adapter* is a file descriptor and *nmserv* is a **scsi_nmserv** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The caller of this ioctl, must allocate a buffer be referenced by the **scsi_id_list** field. In addition the caller must set the **list_len** field to indicate the size of the buffer in bytes.

On successful completion, the **num_ids** field indicates the number of SCSI IDs returned in the current list. If the more ids were found then could be placed in the list, then the adapter driver updates the **list_len** field to indicate the length of buffer needed to receive all SCSI IDs.

Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | A system error has occurred. Consider retrying the operation several times, because another attempt might be successful. |
| EFAULT | A user process copy has failed. |
| EINVAL | The physical configuration does not support this request. |
| ENOMEM | A memory request has failed. |
| ETIMEDOUT | The command has timed out. Consider retrying the operation several times, because another attempt might be successful. |
| ENODEV | The device is not responding. Possibly no LUNs exist on the present target. |

### *SCIOLQWWN*

This operation issues a request to find the SCSI ID of a device for the specified world wide name.

**Note: SCIOLQWWN** is specific to the FCP transport type.

The following is a typical call:

```
rc = ioctl(adapter, SCIOLQWWN, &qrywwn);
```

where *adapter* is a file descriptor and *qrywwn* is a **scsi_qry_wwn** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The caller of this ioctl, must specify the device's world wide name in the **world_wide_name** field. On successful completion, the **scsi_id** field is returned with the SCSI ID of the device with this world wide name.

Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | A system error has occurred. Consider retrying the operation several times, because another attempt might be successful. |
| EFAULT | A user process copy has failed. |
| EINVAL | The physical configuration does not support this request. |
| ENOMEM | A memory request has failed. |

| Item | Description |
|---|---|
| ETIMEDOUT | The command has timed out. Consider retrying the operation several times, because another attempt might be successful. |
| ENODEV | The device is not responding. Possibly no LUNs exist on the present FCP ID. |

### *SCIOLPAYLD*

The **SCIOLPAYLD** ioctl is not supported by the Virtual SCSI adapter driver.

**Note:** The **SCIOLPAYLD** operation is specific to the iSCSI and FCP transport types.

This operation provides the means for issuing a transport payload to the specified device. The SAM adapter driver performs no error recovery or logging on failures of this ioctl operation. The following is a typical call:

```
rc = ioctl(adapter, SCIOLPAYLD, &payld);
```

where *adapter* is a file descriptor and *payld* is a **scsi_trans_payld** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The SCSI ID or SCSI ID alias should be placed in the **scsi_trans_payld**. In addition the user must allocate a payload buffer referenced by the **payld_buffer**field and a response buffer referenced by the **response_buffer** field. The fields **payld_size** and **response_size** specify the size in bytes of the payload buffer and response buffer, respectively. In addition the caller can also set **payld_type** (for FC this is the FC-4 type), and **payld_ctl** (for FC this is the router control field),.

If the **SCIOLPAYLD** operation returns a value of -1 and the **errno** global variable is set to a nonzero value, the requested operation has failed. In this case, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken.

Possible **errno** values are:

| Item | Description |
|---|---|
| EIO | A system error has occurred. |
| EFAULT | A user process copy has failed. |
| EINVAL | Payload and or response buffer are too large. For FCP, iSCSI, and Virtual SCSI Client the maximum size is 4096 bytes. |
| ENOMEM | A memory request has failed. |
| ETIMEDOUT | The command has timed out. Consider retrying the operation several times, because another attempt might be successful. |
| ENODEV | The device is not responding. |
| ETIMEDOUT | The operation did not complete before the time-out value was exceeded. |

### *SCIOLCHBA*

The **SCIOLCHBA** ioctl is not supported by the Virtual SCSI adapter driver.

**Note:** The **SCIOLCHBA** operation is specific to the iSCSI and FCP transport types.

When the device has been successfully opened, the **SCIOLCHBA** operation provides the means for issuing one or more common host bus adapter (HBA) API commands to the adapter. The SAM adapter driver performs full error recovery on failures of this operation.

The **arg** parameter contains the address of a **scsi_chba** structure, which is defined in the **/usr/include/sys/scsi_buf.h** file.

The **cmd** field in the **scsi_chba** structure determines the common HBA API operation that is performed.

If the **SCIOLCHBA** operation fails, the subroutine returns a value of -1 and sets the errno global variable to a nonzero value. In this case, the caller should evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions should be taken.

If a **SCIOLCHBA** operation fails because a field in the **scsi_chba** structure has an invalid value, the subroutine returns a value of -1 and set the errno global variable to EINVAL.

### *SCIOLPASSTHRUHBA*

The **SCIOLPASSTHRUHBA** ioctl is not supported by the Virtual SCSI adapter driver.

**Note:** The **SCIOLPASSTHRUHBA** operation is specific to the iSCSI and FCP transport types.

When the device has been successfully opened, the **SCIOLPASSTHRUHBA** operation provides the means for issuing **passthru** commands to the adapter. The SAM adapter driver performs full error recovery on failures of this operation.

The **arg** parameter contains the address of a *scsi_passthru_hba* structure, which is defined in the **/usr/include/sys/scsi_buf.h** file.

The **cmd** field in the **scsi_passthru_hba** structure determines the type of **passthru** operation to be performed.

If the **SCIOLPASSTHRUHBA** operation fails, the subroutine returns a value of -1 and sets the errno global variable to a nonzero value. In this case, the caller should evaluate the returned status bytes to determine why the operation was unsuccessful and what recovery actions should be taken.

If a **SCIOLPASSTHRUHBA** operation fails because a field in the **scsi_passthru_hba** structure has an invalid value, the subroutine returns a value of -1 and set the errno global variable to EINVAL.

## SAM Subsystem Overview

This section frequently refers to both *device driver* and *adapter device driver*. These two distinct device drivers work together in a layered approach to support attachment of a range of devices.

The adapter device driver is the *lower* device driver of the pair, and the device driver is the *upper* device driver.

### Responsibilities of the Adapter Device Driver

The adapter device driver is the software interface to the system hardware. This hardware includes the transport layer hardware, plus any other system I/O hardware required to run an I/O request. The adapter device driver hides the details of the I/O hardware from the device driver. The design of the software interface lets a user with limited knowledge of the system hardware write the upper device driver.

The adapter device driver manages the transport layer but not the devices. It can send and receive commands, but it cannot interpret the contents of the command. The lower driver also provides recovery and logging for errors related to the transport layer and system I/O hardware. Management of the device specifics is left to the device driver. The interface of the two drivers supports communication between the upper driver and the different transport layer adapters without requiring special code paths for each adapter.

### Responsibilities of the Device Driver

The device driver provides the rest of the operating system with the software interface to a given device or device class. The upper layer recognizes which commands are required to control a particular device or device class. The device driver builds I/O requests containing device commands, and sends them to the adapter device driver in the sequence needed to operate the device successfully. The device driver cannot manage adapter resources or give the command to the adapter. Specifics about the adapter and system hardware are left to the lower layer.

The device driver also provides recovery and logging for errors related to the device that it controls.

The operating system provides several kernel services that let the device driver communicate with adapter device driver entry points without having the actual name or address of those entry points. See “Logical File System Kernel Services” on page 65 for more information.

## Communication between Devices

When two devices communicate, one assumes the initiator-mode role, and the other assumes the target-mode role.

The initiator-mode device generates the command, which requests an operation, and the target-mode device receives the command and acts. It is possible for a device to perform both roles simultaneously.

When writing a new adapter device driver, the writer must know which mode or modes must be supported to meet the requirements of the adapter and any interfaced device drivers.

## Initiator-Mode Support

Communication between the device driver and the adapter device driver for a particular initiator I/O request is made through the **scsi_buf** structure, which is passed to and from the **strategy** subroutine in the same way a standard driver uses a **struct buf** structure.

The interface between the device driver and the adapter device driver for initiator-mode support (that is, the attached device acts as a target) is accessed through calls to the adapter device driver **open**, **close**, **ioctl**, and **strategy** subroutines. I/O requests are queued to the adapter device driver through calls to its strategy entry point.

## Fast I/O Failure for Fibre Channel Devices

AIX supports Fast I/O Failure for Fibre Channel (FC) devices after link events in a switched environment. If the FC adapter driver detects a link event, such as a lost link between a storage device and a switch, the FC adapter driver waits a short period of time, approximately 15 seconds, so that the fabric can stabilize. At that point, if the FC adapter driver detects that the device is not on the fabric, it begins failing all I/Os at the adapter driver. Any new I/O or future retries of the failed I/Os are failed immediately by the adapter until the adapter driver detects that the device has rejoined the fabric.

Fast Failure of I/O is controlled by a new `fscsi` device attribute, `fc_err_recov`. The default setting for this attribute is `delayed_fail`, which is the I/O failure behavior seen in previous versions of AIX. To enable Fast I/O Failure, set this attribute to `fast_fail`, as shown in the example:

```
chdev -l fscsi0 -a fc_err_recov=fast_fail
```

In this example, the `fscsi` device instance is `fscsi0`. Fast fail logic is called when the adapter driver receives an indication from the switch that there has been a link event involving a remote storage device port by way of a Registered State Change Notification (RSCN) from the switch.

Fast I/O Failure is useful in situations where multipathing software is used. Setting the `fc_err_recov` attribute to `fast_fail` can decrease the I/O fail times because of link loss between the storage device and switch. This would support faster failover to alternate paths.

In single-path configurations, especially configurations with a single path to a paging device, the `delayed_fail` default setting is recommended.

Fast I/O Failure requires the following:

- A switched environment. It is not supported in arbitrated loop environments, including public loop.
- FC 6227 adapter firmware, level 3.22A1 or higher.
- FC 6228 adapter firmware, level 3.82A1 or higher.
- FC 6239 adapter firmware, all levels.
- All subsequent FC adapter releases support Fast I/O Failure.

If any of these requirements is not met, the `fscsi` device logs an error log of type `INFO` indicating that one of these requirements is not met and that Fast I/O Failure is not enabled.

## Dynamic Tracking of Fibre Channel Devices

AIX supports dynamic tracking of Fibre Channel (FC) devices. Previous releases of AIX required a user to unconfigure FC storage device and adapter device instances before making changes on the system area network (SAN) that might result in an N_Port ID (SCSI ID) change of any remote storage ports.

If dynamic tracking of FC devices is enabled, the FC adapter driver detects when the Fibre Channel N_Port ID of a device changes. The FC adapter driver then reroutes traffic destined for that device to the new address while the devices are still online. Events that can cause an N_Port ID to change include moving a cable between a switch and storage device from one switch port to another, connecting two separate switches using an inter-switch link (ISL), and possibly rebooting a switch.

Dynamic tracking of FC devices is controlled by a new `fscsi` device attribute, `dyntrk`. The default setting for this attribute is no. To enable dynamic tracking of FC devices, set this attribute to `dyntrk=yes`, as shown in the example:

```
chdev -l fscsi0 -a dyntrk=yes
```

In this example, the `fscsi` device instance is `fscsi0`. Dynamic tracking logic is called when the adapter driver receives an indication from the switch that there has been a link event involving a remote storage device port.

Dynamic tracking support requires the following:

- A switched environment. It is not supported in arbitrated loop environments, including public loop.
- FC 6227 adapter firmware, level 3.22A1 or higher.
- FC 6228 adapter firmware, level 3.82A1 or higher.
- FC 6239 adapter firmware, all levels.
- All subsequent FC adapter releases support Fast I/O Failure.
- The World Wide Name (Port Name) and Node Names devices must remain constant, and the World Wide Name device must be unique. Changing the World Wide Name or Node Name of an available or online device can result in I/O failures. In addition, each FC storage device instance must have `world_wide_name` and `node_name` attributes. Updated filesets that contain the `sn_location` attribute (see the following bullet) must also be updated to contain both of these attributes.
- The storage device must provide a reliable method to extract a unique serial number for each LUN. The AIX® FC device drivers do not autodetect serial number location, so the method for serial number extraction must be explicitly provided by any storage vendor in order to support dynamic tracking for their devices. This information is conveyed to the drivers using the `sn_location` ODM attribute for each storage device. If the disk or tape driver detects that the `sn_location` ODM attribute is missing, an error log of type `INFO` is generated and dynamic tracking is not enabled.

  **Note:** The `sn_location` attribute might not be displayed, so running the `lsattr` command on an hdisk, for example, might not show the attribute even though it could be present in ODM.
- The FC device drivers can track devices on a *SAN fabric*, which is a fabric as seen from a single host bus adapter, if the N_Port IDs on the fabric stabilize within about 15 seconds. If cables are not reseated or N_Port IDs continue to change after the initial 15 seconds, I/O failures could result.
- Devices are not tracked across host bus adapters. Devices only track if they remain visible from the same HBA that they were originally connected to. For example, if device A is moved from one location to another on fabric A attached to host bus adapter A (in other words, its N_Port on fabric A changes), the device is seamlessly tracked without any user intervention, and I/O to this device can continue. However, if a device A is visible from HBA A but not from HBA B, and device A is moved from the fabric attached to HBA A to the fabric attached to HBA B, device A is not accessible on fabric A nor on fabric B. User intervention would be required to make it available on fabric B by running the `cfgmgr` command. The AIX® device instance on fabric A would no longer be usable, and a new device instance on fabric

B would be created. This device would have to be added manually to volume groups, multipath device instances, and so on. This is essentially the same as removing a device from fabric A and adding a new device to fabric B.

- No dynamic tracking can be performed for FC dump devices while an AIX® system dump is in progress. In addition, dynamic tracking is not supported when booting or running the `cfgmgr` command. SAN changes should not be made while any of these operations are in progress.
- After devices are tracked, ODM might contain stale information because Small Computer System Interface (SCSI) IDs in ODM no longer reflect actual SCSI IDs on the SAN. ODM remains in this state until `cfgmgr` is run manually or the system is rebooted (provided all drivers, including any third party FC SCSI target drivers, are dynamic-tracking capable). If `cfgmgr` is run manually, `cfgmgr` must be run on all affected `fscsi` devices. This can be accomplished by running `cfgmgr` without any options, or by running `cfgmgr` on each `fscsi` device individually.

  **Note:** Running `cfgmgr` at run time to recalibrate the SCSI IDs might not update the SCSI ID in ODM for a storage device if the storage device is currently opened, such as when volume groups are varied on. The `cfgmgr` command must be run on devices that are not opened or the system must be rebooted to recalibrate the SCSI IDs. Stale SCSI IDs in ODM have no adverse affect on the FC drivers, and recalibration of SCSI IDs in ODM is not necessary for the FC drivers to function properly. Any applications that communicate with the adapter driver directly using `ioctl` calls and that use the SCSI ID values from ODM, however, must be updated (see the next bullet) to avoid using potentially stale SCSI IDs.
- All applications and kernel extensions that communicate with the FC adapter driver, either through `ioctl` calls or directly to the FC driver's entry points, must support the version `1` `ioctl` and `scsi_buf` APIs of the FC adapter driver to work properly with FC dynamic tracking. Noncompliant applications or kernel extensions might not function properly or might even fail after a dynamic tracking event. If the FC adapter driver detects an application or kernel extension that is not adhering to the new version `1` `ioctl` and `scsi_buf` API, an error log of type `INF0` is generated and dynamic tracking might not be enabled for the device that this application or kernel extension is trying to communicate with. For ISVs developing kernel extensions or applications that communicate with the AIX® Fibre Channel Driver stack, refer to the “SAM Adapter Device Driver ioctl Commands” on page 313 and “Understanding the scsi_buf Structure” on page 303 for changes necessary to support dynamic tracking.
- Even with dynamic tracking enabled, users should make SAN changes, such as moving or swapping cables and establishing ISL links, during maintenance windows. Making SAN changes during full production runs is discouraged because the interval of time to perform any SAN changes is too short. Cables that are not reseated correctly, for example, could result in I/O failures. Performing these operations during periods of little or no traffic minimizes the impact of I/O failures.

The base AIX® FC SCSI Disk and FC SCSI Tape and FastT device drivers support dynamic tracking. The IBM® ESS, EMC Symmetrix, and HDS storage devices support dynamic tracking provided that the vendor provides the ODM filesets with the necessary `sn_location` and `node_name` attributes. Contact the storage vendor if you are not sure if your current level of ODM fileset supports dynamic tracking.

If vendor-specific ODM entries are not being used for the storage device, but the ESS, Symmetrix, or HDS storage subsystem is configured with the `MPIO Other FC SCSI Disk` message, dynamic tracking is supported for the devices in this configuration. This supersedes the need for the `sn_location` attribute. All current AIX® Path Control Modules (PCM) shipped with the AIX® base support dynamic tracking.

The STK tape device using the standard AIX® device driver also supports dynamic tracking provided the STK fileset contains the necessary `sn_location` and `node_name` attributes.

**Note:** SAN changes involving tape devices should be made with no active I/O. Because of the serial nature of tape devices, a single I/O failure can cause an application to fail, including tape backups.

Devices configured with the `Other FC SCSI Disk` or `Other FC SCSI Tape` messages do not support dynamic tracking.

## Fast I/O Failure and Dynamic Tracking Interaction

Although Fast I/O Failure and dynamic tracking of FC Devices are technically separate features, the enabling of one can change the interpretation of the other in certain situations.

The following table shows the behavior exhibited by the FC drivers with the various permutations of these settings:

| dyntrk | fc_err_recov | FC Driver Behavior |
|---|---|---|
| no | `delayed_fail` | The default setting. This is legacy behavior existing in previous versions of AIX®. The FC drivers do not recover if the SCSI ID of a device changes, and I/Os take longer to fail when a link loss occurs between a remote storage port and switch. This might be preferable in single-path situations if dynamic tracking support is not a requirement. |
| no | `fast_fail` | If the driver receives a RSCN from the switch, this could indicate a link loss between a remote storage port and switch. After an initial 15-second delay, the FC drivers query to see if the device is on the fabric. If not, I/Os are flushed back by the adapter. Future retries or new I/Os fail immediately if the device is still not on the fabric. If the FC drivers detect that the device is on the fabric but the SCSI ID has changed, the FC device drivers do not recover, and the I/Os fail with PERM errors. |
| yes | `delayed_fail` | If the driver receives a RSCN from the switch, this could indicate a link loss between a remote storage port and switch. After an initial 15-second delay, the FC drivers query to see if the device is on the fabric. If not, I/Os are flushed back by the adapter. Future retries or new I/Os fail immediately if the device is still not on the fabric, although the storage driver (disk, tape, FastT) drivers might inject a small delay (2-5 seconds) between I/O retries. If the FC drivers detect that the device is on the fabric but the SCSI ID has changed, the FC device drivers reroute traffic to the new SCSI ID. |

| dyntrk | fc_err_recov | FC Driver Behavior |
|---|---|---|
| yes | fast_fail | If the driver receives a Registered State Change Notification (RSCN) from the switch, this could indicate a link loss between a remote storage port and switch. After an initial 15-second delay, the FC drivers query to see if the device is on the fabric. If not, I/Os are flushed back by the adapter. Future retries or new I/Os fail immediately if the device is still not on the fabric. The storage driver (disk, tape, FastT) will likely not delay between retries. If the FC drivers detect the device is on the fabric but the SCSI ID has changed, the FC device drivers reroute traffic to the new SCSI ID. |

When dynamic tracking is disabled, there is a marked difference between the `delayed_fail` and `fast_fail` settings of the `fc_err_recov` attribute. However, with dynamic tracking enabled, the setting of the `fc_err_recov` attribute is less significant. This is because there is some overlap in the dynamic tracking and fast fail error-recovery policies. Therefore, enabling dynamic tracking inherently enables some of the fast fail logic.

The general error recovery procedure when a device is no longer reachable on the fabric is the same for both `fc_err_recov` settings with dynamic tracking enabled. The minor difference is that the storage drivers can choose to inject delays between I/O retries if `fc_err_recov` is set to `delayed_fail`. This increases the I/O failure time by an additional amount, depending on the delay value and number of retries, before permanently failing the I/O. With high I/O traffic, however, the difference between `delayed_fail` and `fast_fail` might be more noticeable.

SAN administrators might want to experiment with these settings to find the correct combination of settings for their environment.

## SAM Asynchronous Event Handling

A device driver can register a particular device instance for receiving asynchronous event status by calling the **SCIOLEVENT** ioctl operation for the adapter device driver.

When an event covered by the **SCIOLEVENT** ioctl operation is detected by the adapter device driver, it builds an **scsi_event_info** structure and passes a pointer to the structure and to the asynchronous event-handler routine entry point, which was previously registered. The fields in the structure are filled in by the adapter device driver as follows:

**async_correlator**
This field is set to the value passed to the adapter device driver in the **scsi_event_struct** structure. The device driver might optionally use this field to provide an efficient means of associating event status with the device instance it goes with. Alternatively, the device driver would use the combination of the **id**, **lun**, **mode**, and **adap_devno** fields to identify the device instance.

**adap_devno**
This field is set to indicate the device major and minor numbers of the adapter on which the device is located.

**events**
This field is set to indicate what event or events are being reported. The following values are possible, as defined in the **src/bos/kernel/sys/scsi_buf.h** file:

**SCSI_FATAL_HDW_ERR**
A fatal adapter hardware error occurred.

**SCSI_ADAP_CMD_FAILED**
An unrecoverable adapter command failure occurred.

**SCSI_RESET_EVENT**
A transport layer reset was detected.

**SCSI_BUFS_EXHAUSTED**
In target-mode, a maximum buffer usage event has occurred.

**SCSI_EVNT_DYNTRK_ON**
For FCP devices, indicates whether dynamic tracking has been enabled (via the **dyntrk** tunable).

**SCSI_EVNT_DYNTRK_OFF**
For FCP devices, indicates whether dynamic tracking has been disabled (via the **dyntrk** tunable).

**SCSI_EVNT_FASTFAIL_ON**
For FCP devices, indicates whether fast fail has been enabled (via the **fc_err_recov** tunable).

**SCSI_EVNT_FASTFAIL_OFF**
For FCP devices, indicattes whether fast fail has been disabled (via the **fc_err_recov** tunable).

**SCSI_EVNT_LUN_MISSING**
A device at the specified **lun_id** field is unreachable.

**SCSI_EVNT_LUN_FOUND**
A device at the specified **lun_id** field was detected.

**SCSI_EVNT_PORT_MISSING**
A remote port at the specified **world_wide_name** field, **scsi_id** field, or both is unreachable.

**SCSI_EVNT_PORT_FOUND**
A remote port at the specified **world_wide_name** field, **scsi_id** field, or both was detected.

**SCSI_EVNT_LINK_DOWN**
The Fibre Channel connectivity to the storage area network (SAN) is temporarily unavailable.

**SCSI_EVNT_LINK_UP**
The Fibre Channel connectivity to the storage area network (SAN) is available.

**SCSI_EVNT_LINK_DEAD**
The Fibre Channel connectivity to the storage area network (SAN) is permanently unavailable.

**SCSI_EVNT_MIGRATED**
The logical partition is migrated.

**SCSI_EVNT_ADAP_ID_CHNG**
The **scsi_id** field of the adapter has changed.

**SCSI_EVNT_TARGET_RST**
A **Target Reset** command is issued to a remote port at the specified **world_wide_name** field, the **scsi_id** field, or both.

**SCSI_EVNT_LUN_RST**
A **LUN reset** command is issued to a device at the specified **lun_id** field.

**SCSI_EVNT_ABORT_TS**
An **Abort Task Set** command is issued.

**lun_id**
For initiator mode, this is set to the SCSI LUN of the attached target device. For target mode, this is set to 0.

**mode**
Identifies whether the initiator or target mode device is being reported. The following values are possible:

**SCSI_IM_MODE**
An initiator mode device is being reported.

**SCSI_TM_MODE**
A target mode device is being reported.

**SCSI_PROMISC_MODE**
All initiator and target mode devices are being reported.

**scsi_id**
For initiator mode, this is set to the SCSI ID or SCSI ID alias of the attached target device. For target mode, this is set to the SCSI ID or SCSI ID alias of the attached initiator device.

The information reported in the **scsi_event_info.events** field does not queue to the device driver, but is instead reported as one or more flags as they occur. Because the data does not queue, the adapter device driver writer can use a single **scsi_event_info** structure and pass it one at a time, by pointer, to each asynchronous event handler routine for the appropriate device instance. After determining for which device the events are being reported, the device driver must copy the **scsi_event_info.events** field into local space and must not modify the contents of the rest of the **scsi_event_info** structure.

Because the event status is optional, the device driver writer determines which action is necessary to take upon receiving event status. The writer might decide to save the status and report it back to the calling application, or the device driver or application level program can take error recovery actions.

**Note:** SAM asynchronous event handling is not supported by all adapter device drivers.

## Defined Events and Recovery Actions

The adapter fatal hardware failure event is intended to indicate that no further commands to or from this device are likely to succeed, because the adapter to which it is attached, has failed. It is recommended that the application end the session with the device.

The unrecoverable adapter command failure event is not necessarily a fatal condition, but it can indicate that the adapter is not functioning properly. Possible actions by the application program include:

- Ending of the session with the device in the near future.
- Ending of the session after multiple (two or more) such events.
- Attempt to continue the session indefinitely.

The SCSI Reset detection event is mainly intended as information only, but can be used by the application to perform further actions, if necessary.

The maximum buffer usage detected event only applies to a given target-mode device; it will not be reported for an initiator-mode device. This event indicates to the application that this particular target-mode device instance has filled its maximum allotted buffer space. The application should perform **read** system calls fast enough to prevent this condition. If this event occurs, data is not lost, but it is delayed to prevent further buffer usage. Data reception will be restored when the application empties enough buffers to continue reasonable operations. The **num_bufs** attribute might need to be increased to help minimize this problem. Also, it is possible that regardless of the number of buffers, the application simply is not processing received data fast enough. This might require some fine tuning of the application's data processing routines.

## Asynchronous Event-Handling Routine

The device driver asynchronous event-handling routine is typically called directly from the hardware interrupt-handling routine for the adapter device driver. The device driver writer must be aware of how this affects the design of the device driver.

Because the event handling routine is running on the hardware interrupt level, the device driver must be careful to limit operations in that routine. Processing should be kept to a minimum. In particular, if any error recovery actions are performed, it is recommended that the event-handling routine set state or status flags only and allow a process level routine to perform the actual operations.

The device driver must be careful to disable interrupts at the correct level in places where the device driver's lower execution priority routines manipulate variables that are also modified by the event-handling routine. To allow the device driver to disable at the correct level, the adapter device driver writer

must provide a configuration database attribute that defines the interrupt class, or priority, it runs on. This attribute must be named **intr_priority** so that the device driver configuration method knows which attribute of the parent adapter to query. The device driver configuration method should then pass this interrupt priority value to the device driver along with other configuration data for the device instance.

The SAM device driver writer must follow any other general system rules for writing a routine that must execute in an interrupt environment. For example, the routine must not attempt to sleep or wait on I/O operations. It can perform wakeups to allow the process level to handle those operations.

Because the device driver copies the information from the **scsi_event_info.events** field on each call to its asynchronous event-handling routine, there is no resource to free and no information that must be passed back later to the adapter device driver.

## SAM Error Recovery

If the device is in initiator mode, the error-recovery process varies depending on whether or not the device is supporting command queuing.

Also, some devices might support NACA=1 error recovery. Thus, error recovery needs to deal with the two following concepts.

### Autosense Data

When a device returns a check condition or command terminated (the **scsi_buf.scsi_status** will have the value of `SC_CHECK_CONDITION` or `SC_COMMAND_TERMINATED`, respectively), it will also return the request sense data.

**Note:** Subsequent commands to the device will clear the request sense data.

If the device driver has specified a valid autosense buffer (**scsi_buf.autosense_length** > `0` and the **scsi_buf.autosense_buffer_ptr** field is not NULL), then the adapter device driver will copy the returned autosense data into the buffer referenced by **scsi_buf.autosense_buffer_ptr**. When this occurs, the adapter device driver will set the **SC_AUTOSENSE_DATA_VALID** flag in the **scsi_buf.adap_set_flags**.

When the device driver receives the SCSI status of check condition or command terminated (the **scsi_buf.scsi_status** will have the value of `SC_CHECK_CONDITION` or `SC_COMMAND_TERMINATED`, respectively), it should then determine if the **SC_AUTOSENSE_DATA_VALID** flag is set in the **scsi_buf.adap_set_flags**. If so then it should process the autosense data and not send a SCSI request sense command.

### NACA=1 error recovery

Some devices support setting the NACA (Normal Auto Contingent Allegiance) bit to a value of one (NACA=1) in the control byte of the SCSI command.

If a device returns a check condition or command terminated (the **scsi_buf.scsi_status** will have the value of `SC_CHECK_CONDITION` or `SC_COMMAND_TERMINATED`, respectively) for a command with NACA=1 set, then the device will require a Clear ACA task management request to clear the error condition on the drive. The device driver can issue a Clear ACA task management request by sending a transaction with the **SC_CLEAR_ACA** flag in the **sc_buf.flags** field. The **SC_CLEAR_ACA** flag can be used in conjunction with the **SC_Q_CLR** and **SC_Q_RESUME** flag in the **sc_buf.flags** to clear or resume the queue of transactions for this device, respectively. For more information, see "Initiator-Mode Recovery During Command Tag Queuing" on page 299.

## SAM Initiator-Mode Recovery When Not Command Tag Queuing

If an error such as a check condition or hardware failure occurs, the transaction active during the error is returned with the **scsi_buf.bufstruct.b_error** field set to `EIO`.

Other transactions in the queue might be returned with the **scsi_buf.bufstruct.b_error** field set to `ENXIO`. If the adapter driver decides not to return other outstanding commands it has queued to it, then the failed transaction will be returned to the device driver with an indication that the queue for this device is not

cleared by setting the **SC_DID_NOT_CLEAR_Q** flag in the **scsi_buf.adap_q_status** field. The device driver should process or recover the condition, rerunning any mode selects or device reservations to recover from this condition properly. After this recovery, it should reschedule the transaction that had the error. In many cases, the device driver only needs to retry the unsuccessful operation.

The adapter device driver should never retry a SCSI command on error after the command has successfully been given to the adapter. The consequences for retrying a command at this point range from minimal to catastrophic, depending upon the type of device. Commands for certain devices cannot be retried immediately after a failure (for example, tapes and other sequential access devices). If such an error occurs, the failed command returns an appropriate error status with an **iodone** call to the device driver for error recovery. Only the device driver that originally issued the command knows if the command can be retried on the device. The adapter device driver must only retry commands that were never successfully transferred to the adapter. In this case, if retries are successful, the **scsi_buf** status should not reflect an error. However, the adapter device driver should perform error logging on the retried condition.

The first transaction passed to the adapter device driver during error recovery must include a special flag. This **SC_RESUME** flag in the **scsi_buf.flags** field must be set to inform the adapter device driver that the device driver has recognized the fatal error and is beginning recovery operations. Any transactions passed to the adapter device driver, after the fatal error occurs and before the **SC_RESUME** transaction is issued, should be flushed; that is, returned with an error type of ENXIO through an **iodone** call.

**Note:** If a device driver continues to pass transactions to the adapter device driver after the adapter device driver has flushed the queue, these transactions are also flushed with an error return of ENXIO through the **iodone** service. This gives the device driver a positive indication of all transactions flushed.

## Initiator-Mode Recovery During Command Tag Queuing

If the device driver is queuing multiple transactions to the device and either a check condition error or a command terminated error occurs, the adapter driver does not clear all transactions in its queues for the device. It returns the failed transaction to the device driver with an indication that the queue for this device is not cleared by setting the **SC_DID_NOT_CLEAR_Q** flag in the **scsi_buf.adap_q_status** field.

The adapter driver halts the queue for this device awaiting error recovery notification from the device driver. The device driver then has three options to recover from this error:

- Send one error recovery command (request sense) to the device.
- Clear the adapter driver's queue for this device.
- Resume the adapter driver's queue for this device.

When the adapter driver's queue is halted, the device drive can get sense data from a device by setting the **SC_RESUME** flag in the **scsi_buf.flags** field and the **SC_NO_Q** flag in **scsi_buf.q_tag_msg** field of the request-sense **scsi_buf**. This action notifies the adapter driver that this is an error-recovery transaction and should be sent to the device while the remainder of the queue for the device remains halted. When the request sense completes, the device driver needs to either clear or resume the adapter driver's queue for this device.

The device driver can notify the adapter driver to clear its halted queue by sending a transaction with the **SC_Q_CLR** flag in the **scsi_buf.flags** field. This transaction must not contain a command because it is cleared from the adapter driver's queue without being sent to the adapter. However, this transaction must have the SCSI ID field ( **scsi_buf.scsi_id**) and the LUN field ( **scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN), respectively. Upon receiving an **SC_Q_CLR** transaction, the adapter driver flushes all transactions for this device and sets their **scsi_buf.bufstruct.b_error** fields to ENXIO. The device driver must wait until the **scsi_buf** with the **SC_Q_CLR** flag set is returned before it resumes issuing transactions. The first transaction sent by the device driver after it receives the returned **SC_Q_CLR** transaction must have the **SC_RESUME** flag set in the **scsi_buf.flags** fields.

If the device driver wants the adapter driver to resume its halted queue, it must send a transaction with the **SC_Q_RESUME** flag set in the **scsi_buf.flags** field. This transaction can contain an actual command, but it is not required. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). If this is the

first transaction issued by the device driver after receiving the error (indicating that the adapter driver's queue is halted),then the **SC_RESUME** flag must be set as well as the **SC_Q_RESUME** flag.

## Analyzing Returned Status

The order of precedence should be followed by device drivers when analyzing the returned status.

1. If the **scsi_buf.bufstruct.b_flags** field has the **B_ERROR** flag set, then an error has occurred and the **scsi_buf.bufstruct.b_error** field contains a valid **errno** value.

   If the **b_error** field contains the ENXIO value, either the command needs to be restarted or it was canceled at the request of the device driver.

   If the **b_error** field contains the EIO value, then either one or no flag is set in the **scsi_buf.status_validity** field. If a flag is set, an error in either the **scsi_status** or **adapter_status** field is the cause.

   If the **status_validity** field is 0, then the **scsi_buf.bufstruct.b_resid** field should be examined to see if the command issued was in error. The **b_resid** field can have a value without an error having occurred. To decide whether an error has occurred, the device driver must evaluate this field with regard to the command being sent and the device being driven.

   If the **SC_CHECK_CONDITION** or **SC_COMMAND_TERMINATED** is set in **scsi_status**, then a device driver must analyze the value of **scsi_buf.adap_set_flags** to determine if autosense data was returned from the device.

   If the **SC_AUTOSENSE_DATA_VALID** flag is set in the **scsi_buf.adap_set_flags** field for a device, then the device returned autosense data in the buffer referenced by **scsi_buf.autosense_buffer_ptr**. In this situation the device driver does not need to issue a SCSI request sense to determine the appropriate error recovery for the devices.

   If the device driver is queuing multiple transactions to the device and if either **SC_CHECK_CONDITION** or **SC_COMMAND_TERMINATED** is set in **scsi_status**, then the value of **scsi_buf.adap_q_status** must be analyzed to determine if the adapter driver has cleared its queue for this device. If the adapter driver has not cleared its queue after an error, then it holds that queue in a halted state.

   If **scsi_buf.adap_q_status** is set to 0, the adapter driver has cleared its queue for this device and any transactions outstanding are flushed back to the device driver with an error of ENXIO.

   If the **SC_DID_NOT_CLEAR_Q** flag is set in the **scsi_buf.adap_q_status** field, the adapter driver has not cleared its queue for this device. When this condition occurs, the adapter driver allows the device driver to send one error recovery transaction (request sense) that has the field **scsi_buf.q_tag_msg** set to SC_NO_Q and the field **scsi_buf.flags** set to SC_RESUME. The device driver can then notify the adapter driver to clear or resume its queue for the device by sending a **SC_Q_CLR** or **SC_Q_RESUME** transaction.

   If the device driver does not queue multiple transactions to the device (that is, the **SC_NO_Q** is set in **scsi_buf.q_tag_msg** ), then the adapter clears its queue on error and sets **scsi_buf.adap_q_status** to 0.
2. If the **scsi_buf.bufstruct.b_flags** field does not have the **B_ERROR** flag set, then no error is being reported. However, the device driver should examine the **b_resid** field to check for cases where less data was transferred than expected. For some commands, this occurrence might not represent an error. The device driver must determine if an error has occurred.

   If a nonzero **b_resid** field does represent an error condition, then the device queue is not halted by the adapter device driver. It is possible for one or more succeeding queued commands to be sent to the adapter (and possibly the device). Recovering from this situation is the responsibility of the device driver.
3. In any of the above cases, if **scsi_buf.bufstruct.b_flags** field has the **B_ERROR** flag set, then the queue of the device in question has been halted. The first **scsi_buf** structure sent to recover the error (or continue operations) must have the **SC_RESUME** bit set in the **scsi_buf.flags** field.

## A Typical Initiator-Mode SAM Driver Transaction Sequence

A simplified sequence of events for a transaction between a device driver and an adapter device driver follows.

In this sequence, routine names preceded by **dd_** are part of the device driver, and those preceded by**scsi_** are part of the adapter device driver.

1. The device driver receives a call to its **dd_strategy** routine; any required internal queuing occurs in this routine. The **dd_strategy** entry point then triggers the operation by calling the **dd_start** entry point. The **dd_start** routine invokes the **scsi_strategy** entry point by calling the **devstrategy** kernel service with the relevant **scsi_buf** structure as a parameter.
2. The **scsi_strategy** entry point initially checks the **scsi_buf** structure for validity. These checks include validating the **devno** field, matching the SCSI ID or the LUN to internal tables for configuration purposes, and validating the request size.
3. Although the adapter device driver cannot reorder transactions, it does perform queue chaining. If no other transactions are pending for the requested device, the **scsi_strategy** routine immediately calls the **scsi_start** routine with the new transaction. If there are other transactions pending, the new transaction is added to the tail of the device chain.
4. At each interrupt, the **scsi_intr** interrupt handler verifies the current status. The adapter device driver fills in the **scsi_buf status_validity** field, updating the **scsi_status** and **adapter_status** fields as required. The adapter device driver also enters the **bufstruct.b_resid** field with the number of bytes not transferred from the request. If all the data was transferred, the **b_resid** field is set to a value of `0`. If the SCSI adapter driver is a adapter driver and autosense data is returned from the device, then the adapter driver will also fill in the **adap_set_flags** and **autosense_buffer_ptr** fields of the **scsi_buf** structure. When a transaction completes, the **scsi_intr** routine causes the **scsi_buf** entry to be removed from the device queue and calls the **iodone** kernel service, passing the just dequeued **scsi_buf** structure for the device as the parameter. The **scsi_start** routine is then called again to process the next transaction on the device queue. The **iodone** kernel service calls the device driver **dd_iodone** entry point, signaling the device driver that the particular transaction has completed.
5. The device driver **dd_iodone** routine investigates the I/O completion codes in the **scsi_buf** status entries and performs error recovery, if required. If the operation completed correctly, the device driver dequeues the original buffer structures. It calls the **iodone** kernel service with the original buffer pointers to notify the originator of the request.

## Understanding the Execution of SAM Initiator I/O Requests

During normal processing, many transactions are queued in the device driver.

As the device driver processes these transactions and passes them to the adapter device driver, the device driver moves them to the in-process queue. When the adapter device driver returns through the **iodone** service with one of these transactions, the device driver either recovers any errors on the transaction or returns using the **iodone** kernel service to the calling level.

The device driver can send only one **scsi_buf** structure per call to the adapter device driver. Thus, the **scsi_buf.bufstruct.av_forw** pointer should be `null` when given to the adapter device driver, which indicates that this is the only request. The device driver can queue multiple **scsi_buf** requests by making multiple calls to the adapter device driver strategy routine.

### Spanned (Consolidated) Commands

Some kernel operations may be composed of sequential operations to a device. For example, if consecutive blocks are written to disk, blocks might or might not be in physically consecutive buffer pool blocks.

To enhance the transport layer performance, the device driver should consolidate multiple queued requests when possible into a single command. To allow the adapter device driver the ability to handle the scatter and gather operations required, the **scsi_buf.bp** should always point to the first **buf** structure entry for the spanned transaction. A null-terminated list of additional **struct buf** entries should be chained

from the first field through the **buf.av_forw** field to give the adapter device driver enough information to perform the DMA scatter and gather operations required. This information must include at least the buffer's starting address, length, and cross-memory descriptor.

The spanned requests should always be for requests in either the read or write direction but not both, since the adapter device driver must be given a single command to handle the requests. The spanned request should always consist of complete I/O requests (including the additional **struct buf** entries). The device driver should not attempt to use partial requests to reach the maximum transfer size.

The maximum transfer size is actually adapter-dependent. The **IOCINFO** ioctl operation can be used to discover the adapter device driver's maximum allowable transfer size.

If a transfer size larger than the supported maximum is attempted, the adapter device driver returns a value of EINVAL in the **scsi_buf.bufstruct.b_error** field.

Due to system hardware requirements, the device driver must consolidate only commands that are memory page-aligned at both their starting and ending addresses. Specifically, this applies to the consolidation of *inner* memory buffers. The ending address of the first buffer and the starting address of all subsequent buffers should be memory page-aligned. However, the starting address of the first memory buffer and the ending address of the last do not need to be aligned so.

The purpose of consolidating transactions is to decrease the number of commands and transport layer phases required to perform the required operation. The time required to maintain the simple chain of **buf** structure entries is significantly less than the overhead of multiple (even two) transport layer transactions.

### Fragmented Commands

Single I/O requests larger than the maximum transfer size must be divided into smaller requests by the device driver.

For calls to a device driver's character I/O (read/write) entry points, the **uphysio** kernel service can be used to break up these requests. For a *fragmented command* such as this, the **scsi_buf.bp** field should be null so that the adapter device driver uses only the information in the **scsi_buf** structure to prepare for the DMA operation.

## SAM Command Tag Queuing

Command tag queuing refers to queuing multiple commands to a device. Queuing to the device can improve performance because the device itself determines the most efficient way to order and process commands. Devices that support command tag queuing can be divided into two classes: those that clear their queues on error and those that do not.

Devices that do not clear their queues on error resume processing of queued commands when the error condition is cleared (either by receiving the next command for NACA=0 error recovery or by receiving a Clear ACA task management command for NACA=1 error recovery). Devices that do clear their queues flush all commands currently outstanding.

Command tag queuing requires the adapter, the device, the device driver, and the adapter driver to support this capability. For a device driver to queue multiple commands to a device (that supports command tag queuing), it must be able to provide at least one of the following values in the **scsi_buf.q_tag_msg**:

- **SC_SIMPLE_Q**
- **SC_HEAD_OF_Q**
- **SC_ORDERED_Q**

The disk device driver and adapter driver do support this capability. This implementation provides some queuing-specific changeable attributes for disks that can queue commands. With this information, the disk device driver attempts to queue to the disk, first by queuing commands to the adapter driver. The adapter driver then queues these commands to the adapter, providing that the adapter supports command tag queuing. If the adapter does not support command tag queuing, then the adapter driver sends only one command at a time to the adapter and so multiple commands are not queued to the disk.

**Note:** This operation might not be supported by all adapter device drivers.

# Understanding the scsi_buf Structure

The **scsi_buf** structure is used for communication between the device driver and the adapter device driver during an initiator I/O request.

This structure is passed to and from the strategy routine in the same way a standard driver uses a **struct buf** structure.

## Fields in the scsi_buf Structure

The **scsi_buf** structure contains certain fields used to pass a command and associated parameters to the adapter device driver. Other fields within this structure are used to pass returned status back to the device driver. The **scsi_buf** structure is defined in the **/usr/include/sys/scsi_buf.h** file.

Fields in the **scsi_buf** structure are used as follows:

- Reserved fields should be set to a value of `0`, except where noted.
- The **bufstruct** field contains a copy of the standard **buf** buffer structure that documents the I/O request. Included in this structure, for example, are the buffer address, byte count, and transfer direction. The **b_work** field in the **buf** structure is reserved for use by the adapter device driver. The current definition of the **buf** structure is in the **/usr/include/sys/buf.h** include file.
- The **bp** field points to the original buffer structure received by the device driver from the caller, if any. This can be a chain of entries in the case of spanned transfers (commands that transfer data from or to more than one system-memory buffer). A `null` pointer indicates a nonspanned transfer. The `null` value specifically tells the adapter device driver that all the information needed to perform the DMA data transfer is contained in the **bufstruct** fields of the **scsi_buf** structure.
- The **scsi_command** field, defined as a **scsi_cmd** structure, contains, for example, the SCSI command length, SCSI command, and a flag variable:
  - The **scsi_length** field is the number of bytes in the actual SCSI command. This is normally 6, 10, 12, or 16 (decimal).
  - The **FCP_flags** field contains the following bit flags:

    **SC_NODISC**
    Do not allow the target to disconnect during this command.

    **SC_ASYNC**
    Do not allow the adapter to negotiate for synchronous transfer to the device.

    During normal use, the **SC_NODISC** bit should not be set. Setting this bit allows a device running commands to monopolize the transport layer. Sometimes it is desirable for a particular device to maintain control of the transport layer once it has successfully arbitrated for it; for instance, when this is the only device on the transport layer or the only device that will be in use. For performance reasons, it might not be desirable to go through selections again to save transport layer overhead on each command.

    Also during normal use, the **SC_ASYNC** bit must not be set. It should be set only in cases where a previous command to the device ended in an unexpected transport free condition. This condition is noted as `SCSI_TRANSPORT_FAULT` in the **adapter_status** field of the **scsi_cmd** structure. Because other errors might also result in the `SCSI_TRANSPORT_FAULT` flag being set, the **SC_ASYNC** bit should only be set on the last retry of the failed command.
  - The **scsi_cdb** structure contains the physical SCSI command block. The 6 to 16 bytes of a single SCSI command are stored in consecutive bytes, with the op code identified individually. The **scsi_cdb** structure contains the following fields:

    **scsi_op_code**
    This field specifies the standard SCSI op code for this command.

**scsi_bytes**
This field contains the remaining command-unique bytes of the command block. The actual number of bytes depends on the value in the **scsi_op_code** field.

- The **timeout_value** field specifies the time-out limit (in seconds) to be used for completion of this command. A timeout value of 0 means no time-out is applied to this I/O request.
- The **status_validity** field contains an output parameter that can have one of the following bit flags as a value:

  **SC_SCSI_ERROR**
  The scsi_status field is valid.

  **SC_ADAPTER_ERROR**
  The adapter_status field is valid.
- The **scsi_status** field in the **scsi_buf** structure is an output parameter that provides valid command completion status when its **status_validity** bit is nonzero. The **scsi_buf.bufstruct.b_error** field should be set to **EIO** any time the **scsi_status** field is valid. Typical status values include:

  **SC_GOOD_STATUS**
  The target successfully completed the command.

  **SC_CHECK_CONDITION**
  The target is reporting an error, exception, or other conditions.

  **SC_BUSY_STATUS**
  The target is currently transporting and cannot accept a command now.

  **SC_RESERVATION_CONFLICT**
  The target is reserved by another initiator and cannot be accessed.

  **SC_COMMAND_TERMINATED**
  The target terminated this command after receiving a terminate I/O process message from the adapter.

  **SC_QUEUE_FULL**
  The target's command queue is full, so this command is returned.

  **SC_ACA_ACTIVE**
  The device has an ACA (auto contingent allegiance) condition that requires a Clear ACA to request to clear it.
- The **adapter_status** field is an output parameter that is valid when its **status_validity** bit is nonzero. The **scsi_buf.bufstruct.b_error** field should be set to **EIO** any time the **adapter_status** field is valid. This field contains generic adapter card status. It is intentionally general in coverage so that it can report error status from any typical adapter.

  If an error is detected while an command is running, and the error prevented the command from actually being sent to the transport layer by the adapter, then the error should be processed or recovered, or both, by the adapter device driver.

  If it is recovered successfully by the adapter device driver, the error is logged, as appropriate, but is not reflected in the **adapter_status** byte. If the error cannot be recovered by the adapter device driver, the appropriate **adapter_status** bit is set and the **scsi_buf** structure is returned to the device driver for further processing.

  If an error is detected after the command was actually sent to the device, then it should be processed or recovered, or both, by the device driver.

  For error logging, the adapter device driver logs transport layer and adapter-related conditions, and the device driver logs device-related errors. In the following description, a capital letter (A) after the error name indicates that the adapter device driver handles error logging. A capital letter (H) indicates that the device driver handles error logging.

  Some of the following error conditions indicate a device failure. Others are transport layer or adapter-related.

**SCSI_HOST_IO_BUS_ERR (A)**
The system I/O transport layer generated or detected an error during a DMA or Programmed I/O (PIO) transfer.

**SCSI_TRANSPORT_FAULT (H)**
The transport protocol or hardware was unsuccessful.

**SCSI_CMD_TIMEOUT (H)**
The command timed out before completion.

**SCSI_NO_DEVICE_RESPONSE (H)**
The target device did not respond to selection phase.

**SCSI_ADAPTER_HDW_FAILURE (A)**
The adapter indicated an onboard hardware failure.

**SCSI_ADAPTER_SFW_FAILURE (A)**
The adapter indicated microcode failure.

**SCSI_FUSE_OR_TERMINAL_PWR (A)**
The adapter indicated a blown terminator fuse or bad termination.

**SCSI_TRANSPORT_RESET (A)**
The adapter indicated the transport layer has been reset.

**SCSI_WW_NAME_CHANGE (A)**
The adapter indicated that the device at this SCSI ID has a new worldwide name. For AIX 5.2, if **Dynamic Tracking of FC Devices** is enabled, the adapter driver has detected a change to the **scsi_id** field for this device and a **scsi_buf** structure with the **SC_DEV_RESTART** flag can be sent to the device. For more information, see **SC_DEV_RESTART**. If support for the concurrent update of dynamic tracking was negotiated at **SCIOLSTART** time (by using **SCSI_CONC_DYNTRK** and **SCSI_CONC_DYNTRK_CONF** flags), the current dynamic tracking state is indicated by the value in the **add_adap_status** field. The value will be 0 if dynamic tracking is currently disabled, or the **SCSI_ADAP_CONC_DYNTRK_ON** flag if it is currently enabled.

An adapter status of **SCSI_WW_NAME_CHANGE** indicates where the SCSI ID-to-WWN mapping has changed when dynamic tracking is enabled and indicated that the worldwide name change for this SCSI ID.

If dynamic tracking is disabled, the FC adapter driver assumes that the SCSI ID-to-WWN mapping cannot change. If a cable is moved from remote target port *A* to target port *B*, and target port *B* assumes the SCSI ID that previously belonged to target port *A*, then from the perspective of the driver with dynamic tracking disabled, the worldwide name at this SCSI ID has changed.

By using dynamic tracking enabled, the general error recovery logic is different. The SCSI ID is considered volatile. Therefore devices are tracked by worldwide name. As such, all queries after events such as those described in the above text, are based on worldwide name. The situation described in the previous paragraph would most likely result in a **SCSI_NO_DEVICE_RESPONSE** status, since it would be determined that the worldwide name of port *A* is not reachable. If a cable connected to port *A* was moved from one switch port to another, the SCSI ID of port *A* on the remote target might change. The FC adapter driver will return **SCSI_WW_NAME_CHANGE** in this case, even though the SCSI ID changed and the worldwide name did not change.

**Note:** When **Dynamic Tracking of FC Devices** is enabled, an adapter status of **SCSI_WW_NAME_CHANGE** might mean that the SCSI ID of a given worldwide name on the fabric has changed, and the worldwide name is not changed.

**SCSI_TRANSPORT_BUSY (A)**
The adapter indicated the transport layer is busy.

**SCSI_TRANSPORT_DEAD (A)**
The adapter indicated the transport layer currently inoperative and is likely to remain this way for an extended time.

The **add_status** field contains additional device status. For devices, this field contains the Response code returned.

- When the device driver queues multiple transactions to a device, the **adap_q_status** field indicates whether or not the adapter driver has cleared its queue for this device after an error has occurred. The flag of SC_DID_NOT CLEAR_Q indicates that the adapter driver has not cleared its queue for this device and that it is in a halted state (so none of the pending queued transactions are sent to the device).
- The **q_tag_msg** field indicates if the adapter can attempt to queue this transaction to the device. This information causes the adapter to fill in the Queue Tag Message Code of the queue tag message for a command. The following values are valid for this field:

  **SC_NO_Q**
  Specifies that the adapter does not send a queue tag message for this command, and so the device does not allow more than one command on its command queue. This value must be used for all commands sent to devices that do not support command tag queuing.

  **SC_SIMPLE_Q**
  Specifies placing this command in the device's command queue. The device determines the order that it executes commands in its queue. The SCSI-2 specification calls this value the "Simple Queue Tag Message".

  **SC_HEAD_OF_Q**
  Specifies placing this command first in the device's command queue. This command does not preempt an active command at the device, but it is run before all other commands in the command queue. The SCSI-2 specification calls this value the "Head of Queue Tag Message".

  **SC_ORDERED_Q**
  Specifies placing this command in the device's command queue. The device processes these commands in the order that they are received. The SCSI-2 specification calls this value the "Ordered Queue Tag Message".

  **SC_ACA_Q**
  Specifies placing this command in the device's command queue, when the device has an ACA (Auto Contingent Allegiance) condition. The SCSI-3 Architecture Model calls this value the "ACA task attribute".

  **Note:** Commands with the value of SC_NO_Q for the **q_tag_msg** field (except for request sense commands) should not be queued to a device whose queue contains a command with another value for **q_tag_msg**. If commands with the SC_NO_Q value (except for request sense) are sent to the device, then the device driver must make sure that no active commands are using different values for **q_tag_msg**. Similarly, the device driver must also make sure that a command with a **q_tag_msg** value of SC_ORDERED_Q, SC_HEAD_Q, or SC_SIMPLE_Q is not sent to a device that has a command with the **q_tag_msg** field of SC_NO_Q.
- The **flags** field contains bit flags sent from the device driver to the adapter device driver. The following flags are defined:

  **SC_CLEAR_ACA**
  When set, means the SCSI adapter driver should issue a Clear ACA task management request for this ID/LUN. This flag should be used in conjunction with either the SC_Q_CLR or SC_Q_RESUME flags to clear or resume the SCSI adapter driver's queue for this device. If neither of these flags is used, then this transaction is treated as if the SC_Q_RESUME flag is also set. The transaction containing the SC_CLEAR_ACA flag setting does not require an actual SCSI command in the **sc_buf**. If this transaction contains a SCSI command then it will be processed depending on whether SC_Q_CLR or SC_Q_RESUME is set.

  This transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). This flag is valid only during error recovery of a check condition or command terminated at a command tag queuing.

  **SC_DELAY_CMD**
  When set, means the adapter device driver should delay sending this command (following a reset or BDR to this device) by at least the number of seconds specified to the adapter device driver in its configuration information. For devices that do not require this function, this flag should not be set.

**SC_DEV_RESTART**
If a **scsi_buf** request fails with a status of **SCSI_WW_NAME_CHANGE**, a **scsi_buf** request with the **SC_DEV_RESTART** flag can be sent if the device driver is dynamic tracking capable.

For AIX 5.2, if **Dynamic Tracking of FC Devices** is enabled, a **scsi_buf** request with **SC_DEV_RESTART** performs a handshake, indicating that the device driver acknowledges the device address change and that the FC adapter driver can proceed with tracking operations. If the **SC_DEV_RESTART** flag is set, the **SC_Q_CLR** flag must also be set. In addition, the scsi command cannot be included in the **scsi_buf** structure. Failure to meet these two criteria results in a failure with a adapter status of **SCSI_ADAPTER_SFW_FAILURE**.

After the **SC_DEV_RESTART** call completes successfully, the device driver performs device validation procedures, such as those performed during an open (Test Unit Ready, Inquiry, Serial Number validation, etc.), in order to confirm the identity of the device after the fabric event.

If an **SC_DEV_RESTART** call fails with any adapter status, the **SC_DEV_RESTART** call can be retried as deemed appropriate by the device driver, because a future retry might succeed.

**SC_LUN_RESET**
When set, means the SCSI adapter driver should issue a Lun Reset task management request for this ID/LUN. This flag should be used in conjunction with ethe **SC_Q_CLR** flag flag.The transaction containing this flag setting does allow an actual command to be sent to the adapter driver. However, this transaction must have the the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). If the transaction containing this flag setting is the first issued by the device driver after it receives an error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set also.

**SC_Q_CLR**
When set, means the adapter driver should clear its transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual command in the **scsi_buf** because it is flushed back to the device driver with the rest of the transactions for this ID/LUN. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). This flag is valid only during error recovery of a check condition or command ended at a command tag queuing device when the SC_DID_NOT_CLR_Q flag is set in the **scsi_buf.adap_q_status field**.

**SC_Q_RESUME**
When set, means that the adapter driver should resume its halted transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual command to be sent to the adapter driver. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). If the transaction containing this flag setting is the first issued by the device driver after it receives an error (indicating that the adapter driver's queue is halted), then the SC_RESUME flag must be set also.

**SC_RESUME**
When set, means the adapter device driver should resume transaction queuing for this ID/LUN. Error recovery is complete after a **SCIOLHALT** operation, check condition, or severe transport error. This flag is used to restart the adapter device driver following a reported error.

**SC_TARGET_RESET**
When set, means the SCSI adapter driver should issue a Target Reset task management request for this ID/LUN. This flag should be used in conjunction with ethe **SC_Q_CLR** flag flag.The transaction containing this flag setting does allow an actual command to be sent to the adapter driver. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) filled in with the device's SCSI ID. If the transaction containing this flag setting is the first issued by the device driver after it receives an error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set also.

- The **dev_flags** field contains additional values sent from the device driver to the adapter device driver. The following values are defined:

**FC_CLASS1**
When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 1 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

**FC_CLASS2**
When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 2 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

**FC_CLASS3**
When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 3 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

**FC_CLASS4**
When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 4 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

- The **add_work** field is reserved for use by the adapter device driver.
- The **adap_set_flags** field contains an output parameter that can have one of the following bit flags as a value:

  **SC_AUTOSENSE_DATA_VALID**
  Autosense data was placed in the autosense buffer referenced by the **autosense_buffer_ptr** field.
- The **autosense_length** field contains the length in bytes of the SCSI device driver's sense buffer, which is referenced via the **autosense_buffer_ptr** field. For devices this field must be non-zero, otherwise the autosense data will be lost.
- The **autosense_buffer_ptr** field contains the address of the SCSI devices driver's autosense buffer for this command. For devices this field must be non-NULL, otherwise the autosense data will be lost.
- The **dev_burst_len** field contains the burst size if this write operation in bytes. This should only be set by the device driver if it h as negotiated with the device and it allows burst of write data without transfer readys. For most operations, this should be set to 0.
- The **scsi_id** field contains the 64-bit SCSI ID for this device. This field must be set for devices.
- The **lun_id field** contains the 64-bit lun ID for this device. This field must be set for devices.
- The **kernext_handle** field contains the pointer returned from the **kernext_handle** field of the **scsi_sciolst** argument for the **SCIOLSTART** ioctl operation. For AIX 5.2, if **Dynamic Tracking of FC Devices** is enabled, the **kernext_handle** field must be set for all scsi_buf calls that are sent to the the adapter driver. Failure to do so results in a failure with an adapter status of **SCSI_ADAPTER_SFW_FAILURE**.
- The **version** field contains the version of this **scsi_buf** structure. Beginning with AIX® 5.2, this field should be set to a value of SCSI_VERSION_1. The **version** field of the **scsi_buf** structure should be consistent with the version of the *scsi_sciolst* argument used for the **SCIOLSTART** ioctl operation.

# Other SAM Design Considerations

The SAM design considerations that are listed in this section must also be taken into account.

## Responsibilities of the Device Driver

SAM device drivers are responsible for the following actions:

- Interfacing with block I/O and logical-volume device-driver code in the operating system.

- Translating I/O requests from the operating system into commands suitable for the particular device. These commands are then given to the adapter device driver for execution.
- Issuing any and all commands to the attached device. The adapter device driver sends no commands except those it is directed to send by the calling device driver.
- Managing device reservations and releases. In the operating system, it is assumed that other initiators might be active on the transport layer. Usually, the device driver establishes a reservation at open time and releases it at close time (except when told to do otherwise through parameters in the device driver interface). Depending upon the type of reservations that are used, it might be necessary for a device driver to monitor conditions such as Unit Attention, which indicates that a change is the state of the previously established reservation.

## Options to the openx Subroutine

Device drivers must support eight defined extended options in their open routine.

Additional extended options to the open are also allowed, but they must not conflict with predefined open options (that is, an **openx** subroutine). The defined extended options are bit flags in the *ext* open parameter. These options can be specified singly or in combination with each other. The required *ext* options are defined in the **/usr/include/sys/scsi.h** header file and can have one of the following values:

**SC_FORCED_OPEN**
Does not honor device reservation-conflict status and can include a target-level device reset.

**SC_RETAIN_RESERVATION**
Does not release device on close and can include a logical unit number (LUN)-level device reset.

**SC_DIAGNOSTIC**
Enters diagnostic mode for this device.

**SC_NO_RESERVE**
Prevents the reservation of the device during an **openx** subroutine call to that device. Allows multiple hosts to share a device.

**SC_SINGLE**
Places the selected device in Exclusive Access mode.

## Using the SC_FORCED_OPEN Option

The **SC_FORCED_OPEN** option forces access to a device by taking action to remove any type of reservation on the device that can inhibit access. The type of action to remove the reservation depends upon the specific type of the established reservation.

You can use a **SCIOLRESET** ioctl to perform a target-level reset of the device. After the action to remove the reservation is completed, other commands are sent as in a normal open. If any of the commands fail due to a reservation conflict, the open registers the failure as an **EBUSY** status. This is also the result if a reservation conflict occurs during a normal open. The device driver must require the caller to have appropriate authority to request the **SC_FORCED_OPEN** option because this request can force a device to drop a reservation. If the caller attempts to initiate this system call without the proper authority, the device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**.

## Using the SC_FORCED_OPEN_LUN Option

The **SC_FORCED_OPEN_LUN** option forces access to a device by taking action to remove any type of reservation on the device that can inhibit access.

The type of action needed to remove the reservation depends on the specific type of reservation established. You can use a**SCIOLRESET** ioctl to perform a LUN level reset of the device. After the action to remove the reservation is completed, other commands are sent as in a normal open. If any of the commands fail due to a reservation conflict, the open registers the failure as an **EBUSY** status. This is also the result if a reservation conflict occurs during a normal open. The device driver must require the caller to have appropriate authority to request the **SC_FORCED_OPEN_LUN** option because this request

can force a device to drop a reservation. If the caller tries to initiate this system call without the proper authority, the device driver returns a value of -1, with the **errno** global variable set to a value of **EPERM**.

## Using the SC_RETAIN_RESERVATION Option

The **SC_RETAIN_RESERVATION** option causes the device driver to hold any established reservation as part of an open during the close of the device. For shared devices (for example, disk or CD-ROM), the device driver must OR together this option for all opens to a given device.

If any caller requests this option, the **close** routine does not issue the release even if other opens to the device do not set **SC_RETAIN_RESERVATION**. The device driver should require the caller to have appropriate authority to request the **SC_RETAIN_RESERVATION** option because this request can allow a program to monopolize a device (for example, if this is a nonshared device). If the caller attempts to initiate this system call without the proper authority, the device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**.

## Using the SC_DIAGNOSTIC Option

The **SC_DIAGNOSTIC** option causes the device driver to enter Diagnostic mode for the given device. This option directs the device driver to perform only minimal operations to open a logical path to the device.

No commands should be sent to the device in the **open** or **close** routine when the device is in Diagnostic mode. One or more ioctl operations should be provided by the device driver to allow the caller to issue commands to the attached device for diagnostic purposes.

The **SC_DIAGNOSTIC** option gives the caller an exclusive open to the selected device. This option requires appropriate authority to run. If the caller attempts to execute this system call without the proper authority, the device driver should return a value of -1, with the **errno** global variable set to a value of EPERM. The **SC_DIAGNOSTIC** option may be executed only if the device is not already opened for normal operation. If this ioctl operation is attempted when the device is already opened, or if an **openx** call with the **SC_DIAGNOSTIC** option is already in progress, a return value of -1 should be passed, with the **errno** global variable set to a value of **EACCES**. Once successfully opened with the **SC_DIAGNOSTIC** flag, the device driver is placed in Diagnostic mode for the selected device.

## Using the SC_NO_RESERVE Option

The **SC_NO_RESERVE** option causes the device driver to prohibit any reservation commands during the open of the device. This facilitates the sharing of the device by multiple hosts.

The device driver should require the caller to have appropriate authority to request the **SC_NO_RESERVE** option, because this request allows other hosts to modify data on the device. If a caller does this kind of request then the caller must ensure data integrity between multiple hosts. If the caller attempts to execute this system call without the proper authority, the device driver should return a value of -1, with the **errno** global variable set to a value of EPERM.

## Using the SC_SINGLE Option

The **SC_SINGLE** option causes the device driver to issue a normal open, but does not allow another caller to issue another open until the first caller has closed the device. This request gives the caller an exclusive open to the selected device.

If this **openx** is attempted when the device is already open, a return value of -1 is passed, with the **errno** global variable set to a value of EBUSY.

Once successfully opened, the device is placed in Exclusive Access mode. If another caller tries to do any type of **open**, a return value of -1 is passed, with the **errno** global variable set to a value of EACCES.

The following table shows how the various combinations of *ext* options should be handled in the device driver.

| EXT OPTIONS openx *ext* option | Device Driver Action | |
|---|---|---|
| | **Open** | **Close** |
| None | Normal | Normal |
| diag | No commands | No commands |
| diag + force | Removes a reservation; otherwise, no commands issued | No commands |
| diag + force + no_reserve | Removes a reservation; otherwise, no commands issued | No commands |
| diag + force + no_reserve + single | Removes a reservation; otherwise, no commands issued | No commands |
| diag + force + retain | Removes a reservation; otherwise, no commands issued | No commands |
| diag + force + retain + no_reserve | Removes a reservation; otherwise, no commands issued | No commands |
| diag + force + retain + no_reserve + single | Removes a reservation; otherwise, no commands issued | No commands |
| diag + force + retain + single | Removes a reservation; otherwise, no commands issued | No commands |
| diag + force + single | Removes a reservation; otherwise, no commands issued | No commands |
| diag + no_reserve | No commands | No commands |
| diag + retain | No commands | No commands |
| diag + retain + no_reserve | No commands | No commands |
| diag + retain + no_reserve + single | No commands | No commands |
| diag + retain + single | No commands | No commands |
| diag + single | No commands | No commands |
| diag + single + no_reserve | No commands | No commands |
| force | Normal, except that a reservation is removed before any commands are issued | Normal |
| force + no_reserve | Normal, except that a reservation is removed before any commands are issued | Normal except no RELEASE |
| force + retain | Normal, except that a reservation is removed before any commands are issued | No RELEASE |
| force + retain + no_reserve | Normal, except that a reservation is removed before any commands are issued | No RELEASE |
| force + retain + no_reserve + single | Normal, except that a reservation is removed before any commands are issued | No RELEASE |

| EXT OPTIONS openx *ext* option | Device Driver Action | |
|---|---|---|
| | **Open** | **Close** |
| force + retain + single | Normal, except that a reservation is removed before any commands are issued | No RELEASE |
| force + single | Normal, except that a reservation is removed before any commands are issued | Normal |
| force + single + no_reserve | Normal, except that a reservation is removed before any commands are issued | No RELEASE |
| no_reserve | No RESERVE | No RELEASE |
| retain | Normal | No RELEASE |
| retain + no_reserve | No RESERVE | No RELEASE |
| retain + single | Normal | No RELEASE |
| retain + single + no_reserve | Normal, except no RESERVE command issued. | No RELEASE |
| single | Normal | Normal |
| single + no_reserve | No RESERVE | No RELEASE |

## Closing the Device

When a device driver is preparing to close a device through the adapter device driver, it must ensure that all transactions are complete.

When the adapter device driver receives a **SCIOLSTOP** ioctl operation and there are pending I/O requests, the ioctl operation does not return until all have completed. New requests received during this time are rejected from the adapter device driver's **ddstrategy** routine.

## Error Processing

It is the responsibility of the device driver to process check conditions and other returned errors properly.

The adapter device driver only passes commands without otherwise processing them and is not responsible for device error recovery.

## Device Driver and Adapter Device Driver Interfaces

The device drivers can have both character (raw) and block special files in the **/dev** directory.

The adapter device driver has only character (raw) special files in the **/dev** directory and has only the **ddconfig**, **ddopen**, **ddclose**, **dddump**, and **ddioctl** entry points available to operating system programs. The **ddread** and **ddwrite** entry points are not implemented.

Internally, the **devsw** table has entry points for the **ddconfig**, **ddopen**, **ddclose**, **dddump**, **ddioctl**, and **ddstrat** routines. The device drivers pass their commands to the adapter device driver by calling the adapter device driver **ddstrat** routine. (This routine is unavailable to other operating system programs due to the lack of a block-device special file.)

Access to the adapter device driver's **ddconfig**, **ddopen**, **ddclose**, **dddump**, **ddioctl**, and **ddstrat** entry points by the device drivers is performed through the kernel services provided. These include such services as **fp_opendev**, **fp_close**, **fp_ioctl**, **devdump**, and **devstrat**.

## Performing Dumps

A device driver must have a **dddump** entry point if it drives a dump device.

A adapter device driver must have a **dddump** entry point if it is used to access a system dump device. Examples of dump devices are disks and tapes.

**Note:** Adapter-device-driver writers should be aware that system services providing interrupt and timer services are unavailable for use in the dump routine. Kernel DMA services are assumed to be available for use by the dump routine. The adapter device driver should be designed to ignore extra **DUMPINIT** and **DUMPSTART** commands to the **dddump** entry point.

The **DUMPQUERY** option should return a minimum transfer size of `0` bytes, and a maximum transfer size equal to the maximum transfer size supported by the adapter device driver.

Calls to the adapter device driver **DUMPWRITE** option should use the *arg* parameter as a pointer to the **scsi_buf** structure to be processed. Using this interface, a **write** command can be executed on a previously started (opened) target device. The *uiop* parameter is ignored by the adapter device driver during the **DUMPWRITE** command. Spanned, or consolidated, commands are not supported using the **DUMPWRITE** option. Gathered **write** commands are also not supported using the **DUMPWRITE** option. No queuing of **scsi_buf** structures is supported during dump processing because the dump routine runs essentially as a subroutine call from the caller's dump routine. Control is returned when the entire **scsi_buf** structure has been processed.

**Note:** Also, both adapter-device-driver and device-driver writers should be aware that any error occurring during the **DUMPWRITE** option is considered unsuccessful. Therefore, no error recovery is employed during the **DUMPWRITE**. Return values from the call to the **dddump** routine indicate the specific nature of the failure.

Successful completion of the selected operation is indicated by a `0` return value to the subroutine. Unsuccessful completion is indicated by a return code set to one of the following values for the **errno** global variable. The various **scsi_buf** status fields, including the **b_error** field, are not set by the adapter device driver at completion of the **DUMPWRITE** command. Error logging is, of necessity, not supported during the dump.

- An **errno** value of `EINVAL` indicates that a request that was not valid passed to the adapter device driver, such as to attempt a **DUMPSTART** command before successfully executing a **DUMPINIT** command.
- An **errno** value of `EIO` indicates that the adapter device driver was unable to complete the command due to a lack of required resources or an I/O error.
- An **errno** value of `ETIMEDOUT` indicates that the adapter did not respond with completion status before the passed command time-out value expired.

# SAM Adapter Device Driver ioctl Commands

Various ioctl operations must be performed for proper operation of the adapter device driver.

The ioctl operations described in the following topics are the primary set of ioctl commands that the adapter device driver must implement to support device drivers. Many of these ioctl operations are relevant to all SAM physical transport types, however, some operations are relevant only to a specific transport type. Other ioctl operations might be required in the adapter device driver to support, for example, system management facilities and diagnostics.

**Attention:** The adapter device driver ioctl operations can only be called from the process level. They cannot be run from a call on any more favored priority levels. Attempting to call them from a more favored priority level can result in the system ending abnormally.

**Related concepts**

SAM Adapter ioctl Operations

This overview lists the SAM adapter ioctl operations.

## Universal Serial Bus (USB) Subsystem

The USB device driver protocol stack for AIX consists of several drivers that communicate with each other.

The following figure illustrates several drivers communicating with each other in a layered fashion.

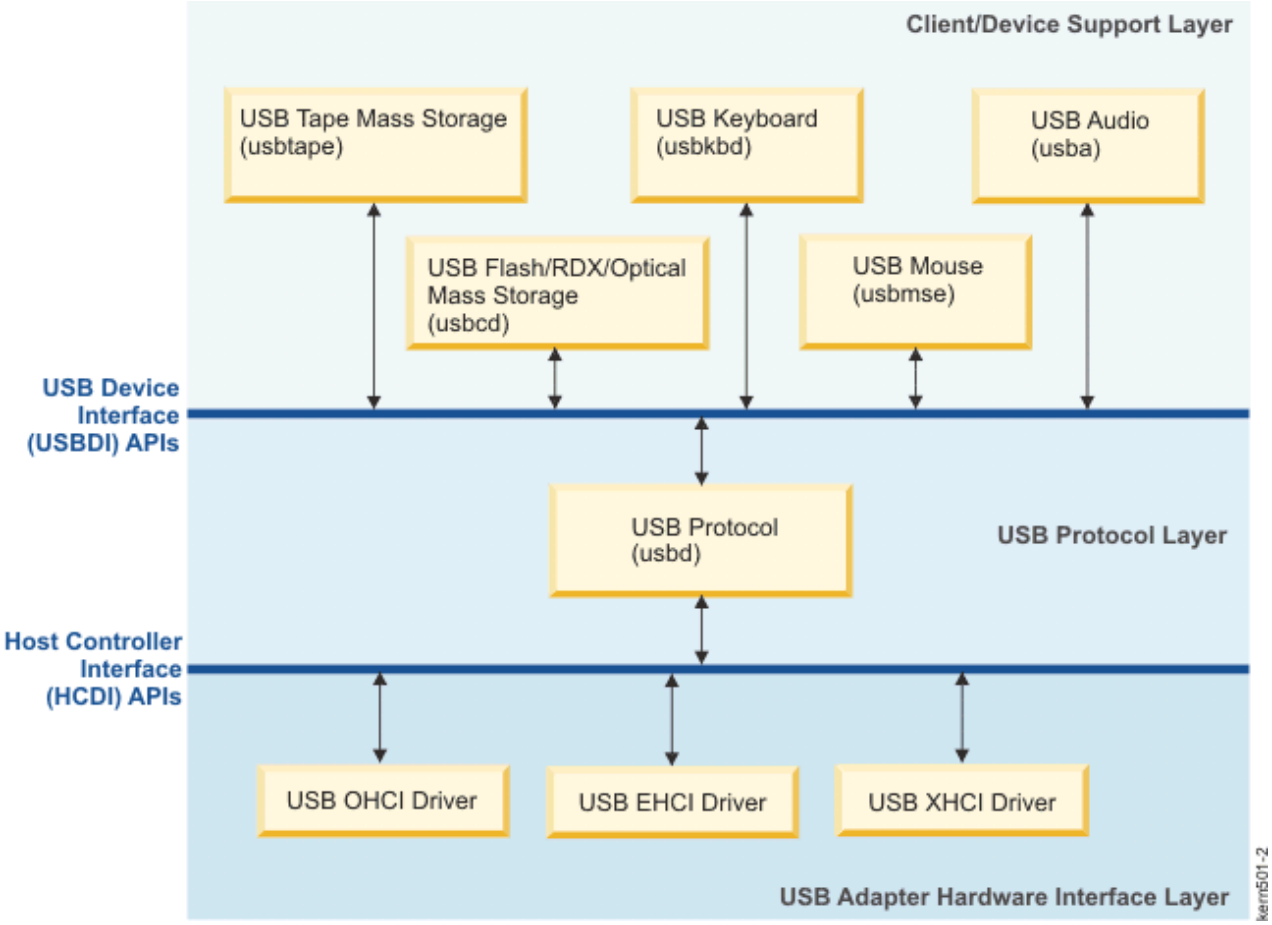


*Figure 1. Universal Serial Bus (USB) Subsystem*

The top layer consists of several client drivers that support various USB classes that include mass storage, human interface device, audio, hub, and other classes. The client drivers hides the class level function from the operating system and provide uniform file-oriented interface to the applications in accessing the corresponding class-level devices. The mid-layer consists of a USB bus driver that hides the host controller implementation details and the bus-level hardware complexity present in the system. It provides uniform interface to each top-level client driver in accessing the corresponding class-level devices regardless of which USB bus the device is on. The lower-layer consists of several host controller drivers that provide the software interface between the host-controller hardware and the USB System Driver (USBD). The details of each host controller driver depend on the host controller interface definition. These three layer drivers in the USB subsystem work together to support the attachment of a range of USB devices. The USB devices such as flash drive, tape, CD-ROM, keyboard, mouse, speaker, and other devices are supported.

The location code is in the [USB Host Controller Number]:[Port Number] format.

For a USB 3.0 controller, 8 logical ports correspond to 4 physical USB ports on the card. The ports are logically numbered based on whether the devices are connected to the port USB 2.0 or to the port USB 3.0. If all devices connected to the ports are USB 2.0 devices, the **lsdev** command displays 1, 2, 3, or 4 for the logical port number. If the devices connected to the port are USB 3.0 devices, the ports are logically numbered as 5, 6, 7, or 8.

The following table shows the logical port values for both **USB 2.0** and **USB 3.0** devices at various physical ports.

| Physical port | Logical port (if USB 2.0 device) | Logical port (if USB 3.0 device) |
|---|---|---|
| T1 (top) | 0.4 | 0.8 |
| T2 | 0.2 | 0.6 |
| T3 | 0.1 | 0.5 |
| T4 | 0.3 | 0.7 |

### Example

```
# lsdev -C | grep usbms
usbms0     Available   0.7         USB Mass Storage
```

As shown above, if a USB 3.0 device (**usbms0**) is showing logical port no = 0.7, then it is connected physically to port T4 of the **usbhc0** controller.

## USB Subsystem Overview

This section refers to all layers that include the USB Client, USB Protocol, and adapter device drivers.

These distinct device drivers work together in a layered approach to support attachment of a range of USB devices.

### Responsibilities of the USB Client Device Driver

The USB client driver or the USB class-specific driver is loaded when the operating system first detects the corresponding class-specific device. It remains loaded until the last device of the corresponding class is removed from the system.

Upon loading, the client driver adds its functions to the kernel device switch table. This step provides the rest of the operating system with the file-oriented software interface to a current USB device class. It registers with the USB system driver (USBD) to determine whether the device is attached and configured. The client driver provides a callback to the USB system driver for attach and disconnect events. Upon detach, the device driver must stop pending IO and return an error to the IO requester. On attachment, the driver must be able to make the device ready to use.

During the operation, the client or class device driver recognizes the file-oriented requests by operating system on a current device and converts them into class-specific requests that the device understands. It uses the uniform call vector interface (USBDI) provided by the USBD to pass these class-specific requests to the device. It is responsible for timing the outstanding requests and performing an error recovery. The error recovery techniques include trying the failed requests again, gathering the sense data, stopping the outstanding requests and resetting the device and the associated pipes.

### Responsibilities of the USB Protocol Device Driver

The USB Protocol driver (USBD) has no hardware that is associated with it. It is configured by rule during system start.

During its configuration, the USB Protocol driver adds its functions to the kernel device switch table. This step provides support for USB configuration methods to register the host controllers that are present in the system. It allows USB client drivers to register callback routines for device connect and disconnect events. It also provides USB client drivers with the uniform pipe-oriented call vectors that allow the client drivers to connect to the device endpoints and perform I/O operation.

During its operation, the USB protocol driver is responsible for following the USB topology behind each registered host controller to detect the USB devices and enumerating them. It reads the descriptors of each detected device during enumeration and initializes the internal data structures. It also handles

Power Budgeting of each USB hub that is detected during the enumeration. It provides an interface to USB client driver to support the various transfer types that includes control, bulk, interrupt, and isochronous through pipe interfaces.

### Responsibilities of the USB Adapter Device Driver

The host controller driver implements the software for a current host controller interface definition. These host controller interface definitions include the Open Host Controller Interface (OHCI), Enhanced Host Controller Interface (EHCI), and eXtensible Host Controller Interface (xHCI).

The host controller driver configures the adapter and provides support for the detection of device attachment and removal. It registers the host controller access routines with the USBD bus driver and provides support for various transfer types: control, bulk, isochronous, and interrupt.

## USB System Device Driver Programming Interface

The USB Protocol Driver (USBD) supports the standard UNIX I/O functions such as `open( )`, `close( )`, and `ioctl( )`. It does not support `read( )`, `write( )`, or `poll( )` functions.

In addition to the standard UNIX interfaces, the driver provides a direct interface by using call vectors. The call vectors are used by a USB client driver to interact with the USBD. The call vectors are started by using the following macros:

| Macros | Description |
|---|---|
| **usbdCloseDevice** | The client driver calls this function to close an open USB logical device. |
| **usbdResetDevice** | The client driver calls this function to reset an open USB logical device. |
| **usbdPipeConnect** | The client driver calls this function to allocate and initialize the resources for a pipe before starting I/O on the pipe. |
| **usbdPipeDisconnect** | The client driver calls this function to close a device. |
| **usbdPipeIO** | The client driver calls this function when it performs an I/O. This function is also called internally by the USB system driver during enumeration, topology discovery, and device configuration. |
| **usbdPipeIOWait** | The client driver and the USB system driver call this function when the caller waits for an outstanding I/O. |
| **usbdPipeStatus** | The client driver calls this function to obtain the status of a pipe. The status of the pipe might be either the pipe is halted or the pipe is active. |
| **usbdPipeAbort** | The client driver calls this function to stop some previously started I/O. If called from an interrupt environment, a request is sent to *config proc* function to stop a specific I/O and the control is returned to the caller immediately. |
| **usbdPipeClear** | The client driver calls this function to activate a previously halted pipe from the host perspective. This service can be started only after issuing **CLEAR FEATURE** command to clear the endpoint halt from the device's perspective. |

| Macros | Description |
| --- | --- |
| **usbdMapMemory** | The client driver uses this function to map host memory so that the host controller can read from or write to the memory by using bus mastered DMA. The USBDMAP_USER flag must be specified whether the memory to be mapped is located in user address space. |
| **usbdUnmapMemory** | The client driver uses this function to unmap previously mapped memory. All I/O referencing the memory block must be completed before this function is called. This function can be called from process environment only. |
| **usbdGetDescriptors** | The client driver uses this function to read the list of descriptors from the device. A pointer to buffer containing the list of descriptors is returned. |
| **usbdGetDevselector** | After reconnect, the *hcdevno* and *addr* values of the device might change from the previous values before disconnect. This can occur when two or more USB logical devices with identical *class*, *subclass*, *protocol*, and *devtype* values are used. This function might be called from process environment only. |
| **usbdGetFrame** | This function is called by the client driver to obtain the current frame number from the host controller to which the device is connected to. This function is used for isochronous pipes only. |
| **usbdSetInterface** | The client driver calls this function when to change an interface to an alternative setting. |

## usbdCloseDevice

### Purpose

Closes an open USB logical device.

### Syntax

```
#include <usbdi.h>
USBstatus usbdCloseDevice(handle)
USBhandle handle;
```

### Parameter

| Item | Description |
| --- | --- |
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |

### Description

The client driver calls this function to close an open USB logical device.

After the function is called, the USB system driver verifies that the passed in client handle is valid and checks if reconnect or disconnect is in progress for the client. If the process of reconnect or disconnect is in progress, the USB system driver returns EAGAIN error to the caller. This step allows the client to loop on EAGAIN error until all the callbacks are complete. It stops all the outstanding I/O on the device, disconnects all the pipes that are associated with the device, and unmaps all the mapped client I/O

buffers. The function then removes the client handle from the list of valid client handles and frees the memory that is associated with the handle.

### Execution Environment

This function might be called from the process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | success |
| EAGAIN | Close failed, devices is being disconnected. All others - failure |

## usbdResetDevice

### Purpose

Resets an open USB logical device.

### Syntax

```
#include <usbdi.h>
USBstatus usbdResetDevice(handle)
USBhandle handle;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |

### Description

The USB client driver calls this function to reset an open USB logical device. The function is completed as the last step of the error recovery procedure. For example, a certain keyboard hub do not assert the port reset signal long enough to properly reset some USB mouse. On these hubs, the port to which the mouse is connected must be reset a second time before the mouse begins to work.

This function returns to the caller immediately. The USBD then proceeds to logically disconnect and reconnect the device that includes making disconnect and reconnect call backs to the client driver if they are enabled. Except for the device handle, this function invalidates all other USB handles such as buffer map handles and pipe handles. These handles must be reacquired after the reconnect call back is made to the client driver.

This function then calls the client driver's disconnect callback routine to free any client-driver level resources that are associated with this device and puts all the luns that is associated with this device in a disconnected state. Finally, the protocol driver logically reconnects the device and attempts to associate the device with the most suitable client. This process is completed by running through all the client handles that are registered with USBD and tries to match the device's class, subclass, and protocol with the client handle. If it finds the matching client handle with this information, it starts the reconnect callback routine that is associated with the client handle. The reconnect callback routine brings back the previously disconnected luns into the available state.

### Execution Environment

This function might be called from interrupt (USBDINTRLEVEL priority or lower) or process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |

## usbdPipeConnect

### Purpose

Connects to a device pipe.

### Syntax

```
#include <usbdi.h>
USBstatus usbdPipeConnect(handle, epaddr, policy, hpipe)

USBhandle handle;
int epaddr;
PIPEPOLICY policy;
USBhandle hpipe;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **epaddr** | Address of the endpoint of the pipe (including direction bit) obtained from endpoint descriptor. |
| **policy** | Pointer to an initialized PIPEPOLICY structure (See **PIPEPOLICY** structure in the **usbdi.h** file).<br>**PIPEPOLICY structure fields**<br>• The **maxNumIRP** field specifies the maximum number of I/O request packets (IRPs) that is enqueued to the pipe at available time.<br>• The **maxPayloadSize** field specifies the maximum number of bytes that are transferred per IRP. A nonzero value is required in this field for all pipes other than the default control pipe.<br>• The **serviceRate** field specifies the maximum allowable time in ms to service an IRP. This field is meaningful only for interrupt and isochronous pipes. |
| **hpipe** | Pointer to location where the USB driver returns the pipe handle. |

### Description

The USB client driver calls this function to connect to a pipe. A logical USB device always has a default control pipe whose address is zero. All other pipes are identified by the device through endpoint descriptors.

This function allocates and initializes the resources for a pipe before starting I/O operations on the pipe. Specifically, the USB protocol driver checks if the pipe associated with the passed in endpoint address is already connected. It calculates the maximum number of IOBs that must be allocated based on the parameters that are passed in the pipe policy data structure. Then, it calls *pipe connect* function of the adapter driver to allocate and start the adapter level resources to support the I/O operation on the pipe.

### Execution Environment

This function might be called from the process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failures |

## usbdPipeDisconnect

### Purpose

Disconnects from a device pipe.

### Syntax

```
#include <usbdi.h>
USBstatus usbdPipeDisconnect(handle, hpipe)
USBhandle handle;
USBhandle hpipe;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **hpipe** | Pipes handle returned from the *usbdPipeConnect* function. |

### Description

The USB client driver calls this function to disconnect from a device pipe. After a pipe is disconnected, the pipe handle is no longer valid and must not be used. Any I/O operations outstanding to and from the pipe is stopped when this function is called. Upon receiving this call, the USB system driver verifies the passed in client and if the pipe handles are valid, the pipe is connected. It calls the *pipe disconnect* function of the adapter driver to free the adapter driver-related data structures.

### Execution Environment

This function might be called from the process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | success |
| All others | failures |

## usbdPipeIO

### Purpose

Initiates I/O process to and from a logical device.

### Syntax

```
#include <usbdi.h>
USBstatus usbdPipeIO(handle, irp)
USBhandle handle;
PIRP irp;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **irp** | Pointer to an initialized I/O request packet (IRP) structure. |

### Description

The USB client driver uses this function to starts USB device I/O. Multiple IRPs for the same USB logical device might be grouped together and presented through one call to this function. The pipe handle in the IRP identifies the pipe and the type of data transfer to be started. The IRPs are enqueued to the logical USB device in the order that they are received from the client driver. The IRPs are posted in the order that they are processed by the pipe that might not be the order in which they were received. The function returns to the caller after the IRPs are enqueued to the pipe.

**Note:** The client is not allowed to change the configuration of the logical USB device. Therefore, certain commands sent to the default control pipe is rejected with a return code of the USBD_ERROR by the USB system driver.

This function is also called internally by the USB protocol driver during enumeration, topology discovery, and device configuration.

Upon receiving this call, the USB Protocol driver breaks down the received IRP into several page sized requests that are called Input/Output Blocks (IOB). It builds the IOBs based on the transfer type that is associated with the pipe. The transfer types such as control, bulk, isochronous, and interrupt are supported. It groups the IOBs and transfers them to adapter driver.

### Execution Environment

This function might be called from the interrupt (USBDINTRLEVEL priority or lower) or process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure (no IRPs delivered) |

## usbdPipeIOWait

### Purpose

Waits until previous I/O request is processed.

### Syntax

```
#include <usbdi.h>
USBstatus usbdPipeIOWait(handle, irp, to)
USBhandle handle;
```

```
PIRP irp;
int to;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **irp** | Pointer to IRP structure used in the *usbdPipeIO* request. |
| **to** | Maximum time to wait for I/O completion that is specified in milliseconds. |

### Description

This function is called both by the client driver and the USB system driver when the call waits for an outstanding I/O. The caller sleeps until the associated I/O request is complete or until the timeout is complete. If the timeout is complete, the outstanding I/O request is stopped and the request is completed with the return status of the USBD_TIMEOUT to the caller.

### Execution Environment

This function might be called from the process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## usbdPipeStatus

### Purpose

Obtains pipe status.

### Syntax

```
#include <usbdi.h>
USBstatus usbdPipeStatus(handle, hpipe, status);
USBhandle handle;
USBhandle hpipe;
PPIPESTATUS status;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **hpipe** | Pipe handles returned by the *usbdPipeConnect* function. |
| **status** | Address of the PIPESTATUS structure where pipe status is returned. |

### Description

The USB client driver calls this function to obtain the status of a pipe. The reported state is either USBD_HALTED (unable to accept or process IRPs) or USBD_ACTIVE (able to accept and process IRPs). It also returns the number of IRPs that are pending on the pipe.

This function does not return the status of the pipe from the device perspective (for example: reflected endpoint state). A USB client driver might obtain this status by sending a USB Standard GetStatus device request which refers to the wanted endpoint address to the default control pipe of the USB device.

### Execution Environment

This function might be called from the interrupt (USBDINTRLEVEL priority or lower) or process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## usbdPipeAbort

### Purpose

Aborts previously started device I/O.

### Syntax

```
#include <usbdi.h>
USBstatus usbdPipeAbort(handle, irp)
USBhandle handle;
PIRP irp;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **irp** | Pointer to the IRP structure used to start the device I/O. |

### Description

The USB client driver uses this function to cancel a previously started I/O request. The function returns after the specified IRP is canceled or processed. The function affects only the specified IRP ignoring the status of any IRPs that might be grouped to the specified IRP. The status of the pipe (halted or active) and the pipe's data synchronization toggle bit is not changed by this function.

### Execution Environment

This function might be called from the process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## usbdPipeClear

### Purpose

Clears halt condition and resume operation of a pipe.

### Syntax

```
#include <usbdi.h>
USBstatus usbdPipeClear(handle, hpipe)
USBhandle handle;
USBhandle hpipe;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **hpipe** | Pipe handles returned by the *usbdPipeConnect* function. |

### Description

The USB client driver uses this function to resume a pipe from the perspective of the host. This command does not affect the reflected endpoint state (for example, pipe status from the perspective of the USB device). If the reflected endpoint state must be changed, a USB_CLEAR_FEATURE control message must be sent to the device that request halted condition of the end point to be cleared before calling the *usbdPipeClear* function. The *usbdPipeClear* function is not available for control and isochronous pipes because they do not halt. The *usbdPipeClear* function does not affect the data toggle synchronization bit.

### Execution Environment

This function might be called from the interrupt (USBDINTRLEVEL priority or lower) or process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## usbdMapMemory

### Purpose

Maps memory so that it can be bus mastered by the host controller.

### Syntax

```
#include <usbdi.h>
USBstatus usbdMapMemory(handle, vaddr, size, flags, hmap)
USBhandle handle;
caddr_t vaddr;
int size;
uint flags;
USBhandle *hmap;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **vaddr** | Virtual address of the memory block. |
| **size** | Size of the memory block in bytes (might cross page boundary). |
| **flags** | The flags are:<br>**USBDMAP_USER**<br>Memory is in user address space.<br>**USBDMAP_PIN**<br>Pin memory before map.<br>**USBDMAP_READ**<br>Buffer to be used to transfer data from device to host. |
| **hmap** | Location to return map handle. |

### Description

The USB client driver uses this function to map host memory so that the host controller might read from or write to the memory by using bus mastered DMA. The USBDMAP_USER flag must be specified whether the memory to be mapped resides in user address space. The owner of the memory must be the process currently active on the processor. The memory is cross memory that is attached, pinned (value of USBDMAP_PIN flag is ignored), and mapped. If USBDMAP_USER is not set, the memory allocated from the kernel's memory heap. If the USBDMAP_PIN flag is specified, the memory is pinned before it is mapped. Otherwise, the memory is already been pinned. The memory block to be mapped might be greater than one physical page in length and might resides in one or more physical pages. The virtual address of the memory block might not be NULL.

### Execution Environment

This function might be called from the process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## usbdUnmapMemory

### Purpose

Unmaps memory.

### Syntax

```
#include <usbdi.h>
USBstatus usbdUnmapMemory(handle, hmap)
USBhandle handle;
USBhandle hmap;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **hmap** | Map handles returned from the *usbdMapMemory* function. |

### Description

The USB client driver uses this function to unmap previously mapped memory. All I/O referencing the memory block must be completed before this function is called. This function can be called from process environment only.

### Execution Environment

This function might be called from the process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## usbdGetDescriptors

### Purpose

Reads the descriptors from the device.

### Syntax

```
#include <usbdi.h>
USBstatus usbdGetDescriptors(handle, ppdesc)
USBhandle handle;
PDESCIDX *ppdesc;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **ppdesc** | Location to store pointer to the descriptor index structure. |

### Description

The USB client driver uses this function to get access to certain common USB descriptors and strings. Upon return from the call, a pointer to a DESCIDX structure is stored in the specified location. The index

array and the descriptors must be treated as read-only data and must not be freed when it is not required. The pointer is not valid after the device is disconnected and must not be used. For more information, see the **DESCIDX** structure defined in the **/usr/include/sys/usbdi.h** file.

### Execution Environment

This function might be called from kernel process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## usbdGetDevselector

### Purpose

Gets the new *devselctor* function after a reconnect process.

### Syntax

```
#include <usbdi.h>
USBstatus get_devselector(handle, pDevSel)
USBhandle handle;
PDEVSELECTOR pDevSel;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **pDevSel** | Location to get the device-specific details such as address, configuration number, and interface number. |

### Description

This function is called by the USB client driver to obtain the current **DEVSELECTOR** values. These values might change because the USB device may disconnect and reconnect. After a reconnect, the **hcdevno** and **addr** fields might change. A USB client driver must know which interfaces have the same **hcdevno** and **addr** values to group interfaces when two or more have the same interface number. This can happen when two or more identical USB devices are being used.

### Execution Environment

This function might be called from kernel process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## usbdGetFrame

### Purpose

Gets the frame number and current speed for isochronous endpoints.

### Syntax

```
#include <usbdi.h>
USBstatus usbdGetDescriptors(handle, pFrame, pSpeed)
USBhandle handle;
uint16_t *pFrame;
uint8_t *pSpeed;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **pFrame** | Location to get the device-specific details such as the address, configuration number, and interface number. |
| **pSpeed** | Location to store current speed. |

### Description

The USB client driver uses this function to obtain the current frame number for the host controller to which the device is connected. This function is used for the isochronous pipes.

If the device is full speed, then this request saves the current frame number at the location that is specified by pFrame and the value USBD_FULL_SPEED at the location that is specified by pSpeed. If the device is high speed, this request saves the current microframe number at the location that is specified by pFrame and the value USBD_HIGH_SPEED at the location that is specified by pSpeed.

The pSpeed pointer might be NULL if the client driver does not need to know the value.

The value USBD_X_Unimplemented (defined in **/usr/include/sys/usbdi.h** file) are returned if isochronous is not supported in the host controller driver to which the device is attached.

### Execution Environment

This function might be called from kernel process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## usbdSetInterface

### Purpose

Used to select an alternative setting of an interface.

### Syntax

```
#include <usbdi.h>
USBstatus usbdSetInterface(handle, alternate)
USBhandle handle;
uint8_t alternate;
```

### Parameter

| Item | Description |
|---|---|
| **handle** | Handle of open USB logical device that is returned by the USBD_OPEN_DEVICE fp_ioctl. |
| **alternate** | The alternative setting number of an interface. |

### Description

The USB client driver uses this function to change an interface to an alternative setting. The client driver must close all pipes to endpoints associated with the handle, except the control pipe, before this call. The USBD verifies that the interface has alternative settings and that the alternative settings have the same *bInterfaceClass*, *bInterfaceSubClass,*, and *bInterfaceProtocol* as the default interface alternative setting 0. Otherwise, it returns an error. The USBD rejects the change for other unknown reasons if it causes problems with device configuration or management. If the current setting is the same as the requested setting, USBD returns USBD_SUCCESS immediately. Then it will verifies if the pipes are closed and return an error if not. It sends the USB_SET_IF request to the interface. If the request returns an error, the USBD tries again for a limited number of times, then it is returned to the client driver. If successful, it frees the existing pipe control blocks, reset the interface descriptor pointer, pIFDesc, and endpoint descriptor pointer, pEPDesc, in the **DESCIDX** structure, and build new pipe control blocks. The USBD returns USBD_SUCCESS to the client driver. The client driver must then call *usbdGetDescriptors* function to get the updated descriptor information.

### Execution Environment

This function might be called from kernel process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## USB Host Controller Driver Programming Interface

The USB Host Controller driver supports the standard file-oriented interface functions such as *open( )*, *close( )*, and *ioctl( )*. In addition to these standard file-oriented interfaces, the driver provides a direct interface through call vectors.

The call vectors are used by a USB System Driver (USBD) to interact with the Host controller driver.

### Host Controller Registration

Whenever the USB configuration method is run, it opens the USBD special file (**/dev/usb0**) and attempts to register each detected and available USB host controller with the USB System Driver through a USBD_REGISTER_HC IOCTL.

During the processing of the USBD_REGISTER_HC IOCTL, the USBD opens the host controller driver and requests for the registration of call vectors that are stored with in the Host controller driver through a

HCD_REGISTER_HC IOCTL. After the call vectors are registered with the USBD, the USB System Driver (USBD) and the Host controller driver communicates through the call vectors.

## Root Hub Emulation

According to the USB architecture, each Host Controller (HC) must implement a USB root hub. The root hub is a pseudo device that gets attached to the root of the USB device tree topology located at the rear of each HC.

The attachment of the root hub starts the device topology discovery process. Since the root hub is a pseudo hub device, the driver for the HC cannot request it to assign a standard USB device address to the root hub. Instead, the root hub is marked as a special hub by setting the `IS_ROOTHUB` flag in the USB System Driver (USBD) internal root hub-related control and interrupt pipe data structures. During the process, the *pipeConnect* function Host controller driver uses this `IS_ROOTHUB` flag to set its root hub-related control and interrupt pipe data structures by using the `emulated` flag. Any operations on the root hub-related control and interrupt pipes are intercepted by the HC and are processed according to the information provided by the Host controller hardware registers.

## HCDI Call Vectors

The Host Controller Device Interface (HCDI) call vectors describes the available functions through call vectors between the USB System Driver and the HCD.

To run a function, the driver calls one of the following macros. To ensure correct serialization with no dead locks, functions that are called within the interrupt environment must be called with a priority of HCD_OFFLEVEL or less. Hence the USB System Driver and Host Controller Drivers perform better by locking by using the priority of HCD_OFFLEVEL instead of INTMAX.

| Macros | Description |
|---|---|
| **hcdUnregisterHC** | This call vector is started to unregister a host controller by using the USBD. |
| **hcdPipeConnect** | This call vector is called to create a pipe to an endpoint on a indicated USB device. |
| **hcdPipeDisconnect** | This call vector is started to remove the previously established pipe connection to the endpoint on a indicated USB device. |
| **hcdPipeIO** | This call vector is started to perform actual I/O operations the device. The I/O operations can be any of the transfer types: control, bulk, isochronous, and interrupt. |
| **usbdPipeIO** | This function is called when the client driver performs an I/O operations. This function is also called internally by the USB system driver during enumeration, topology discovery, and device configuration. |
| **hcdPipeStatus** | This call vector is started by USBD to obtain the status of the pipe from the host perspective. |
| **hcdPipeAbort** | This call vector is started by USBD to cancel the processing of an Input/ Output Block (IOB). The pipe that is specified by the IOB be halted before *hcdPipeAbort* fucntion is called. |
| **hcdPipeHalt** | This call vector is started by USBD to halt a pipe from the perspective of the host controller. All pending I/O operations remains pending. |
| **hcdPipeClear** | This call vector is started to clear, unhalt, and restart the I/O operations on a current endpoint. Upon invocation, the function checks whether the ring is in halted state. |
| **hcdPipeResetToggle** | This call vector is started by USBD to reset the data synchronization toggle bit to DATA0. |
| **hcdPipeAddIOB** | This call vector is started by USBD to increase the maximum number of outstanding I/O buffers. |

| Macros | Description |
|---|---|
| **hcdShutdownComplete** | This call vector is started by USBD to inform the HCD that the *usbdReqHCshutdown* request is completed. |
| **hcdGetFrame** | This call vector is started by USBD to obtain the current frame number from the host controller that the device is connected to. |
| **hcdDevAlloc** | This call vector is started by USBD when it detects the attachment of a USB logical device. |
| **hcdDevFree** | This call vector is started by USBD when it detects the removal of a USB logical device. |
| **hcdConfigPipes** | This call vector is started by USBD during the enumeration of USB logical device. |
| **hcdUnconfigPipes** | This call vector is started by USBD system driver when it detects that a device is removed from the system. |
| **usbdBusMap** | This call vector is started by the adapter driver (xHCD) to map memory for bus mastering by the host controller. |
| **usbdReqHCunregister** | This call vector is started during the removal of Host Controller. The *CFG TERM* function of the adapter driver requests USBD to unregister this Host Controller. |
| **usbdPostIOB** | This call vector is started by the adapter driver to retire an IOB. |
| **usbdReqHCshutdown** | This call vector is started during the removal of Host Controller. |
| **usbdReqHCrestart** | This call is started when an error is detected with the adapter and the recovery of adapter driver from this error requires restarting the adapter. |

### *hcdUnregisterHC*

## Purpose

Closes the communication channel between the USBD and a specific host controller.

## Syntax

```
#include <hcdi.h>
USBstatus hcdUnregisterHC(pHCDI)
PHCDI pHCDI;
```

## Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |

## Description

This function is an HCD supplied function that is called by the USBD when the USBD is unconfigured. It closes the communication channel that is opened through the HCD_REGISTER_HC fp_ioctl. All pipes that are associated with the host controller are disconnected before this vector call is made.

## Execution Environment

This function can only be called from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdPipeConnect*

### Purpose

Connects to a device pipe.

### Syntax

```
#include <hcdi.h>
USBstatus hcdPipeConnect(pHCDI, pPipe, phpipe)
PHCDI pHCDI;
PPIPEDESCRIPTOR pPipe;
USBhandle *phpipe;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **pPipe** | Pointer to start PIPEDESCRIPTOR. |
| **phpipe** | Pointer to location to return pipe handle. |

### Description

This function is an HCD supplied function that is called by the USBD when it wants to connect to a pipe (for example, end point) on a current USB device. The HCD must use the information that is provided in the PIPEDESCRIPTOR structure when setting up the pipe. The structure contains items such as the device address, connection speed, end-point address, and end-point type (For example, control, bulk, interrupt, or isochronous). The HCD might use the PIPEDESCRIPTOR structure directly or copy the information to another location. The USBD does not change the contents of the descriptor until the pipe is connected. After the *hcdPipeConnect* function completes successfully, the pipe must be in an active state and ready to receive and process I/O requests. The Data toggle bit must be initialized to Data0.

### Execution Environment

This function can be from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdPipeDisconnect*

### Purpose

Disconnects from a device pipe.

### Syntax

```
#include <hcdi.h>
USBstatus hcdPipeDisconnect(pHCDI, hpipe)
PHCDI pHCDI;
USBhandle hpipe;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **phpipe** | Pipe handles returned by *hcdConnectPipe* function. |

### Description

This function is an HCD supplied function that is called by the USBD to disconnect from a device pipe. The function is called only after all the I/O operations on the specified pipe is completed or is aborted.

### Execution Environment

This function can only be called from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdPipeIO*

### Purpose

Initiates I/O operations to and from a logical device.

### Syntax

```
#include <hcdi.h>
USBstatus hcdPipeIO(pHCDI, pIOB)
PHCDI pHCDI;
PIOB pIOB;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **pIOB** | Pointer to an initialized IOB structure. |

### Description

This function is an HCD supplied function, which is called by the USBD to initiate device I/O operations. The I/O operations can be any of the following transfer types: control, bulk, isochronous, and interrupt. The USBD presents to the HCD a chain of one or more IOBs with each call to the *hcdPipeIO* function. The pipe handle in each IOB identifies the pipe that must process the IOB. A chain or sub chain of IOBs that reference the same pipe handle must be enqueued to the pipe by the HCD at the same time so that they

are not interrupted by other I/O request. The HCD must enqueue the IOBs even when the pipe is halted. The HCD must return to the USBD after the entire IOB chain is traversed and the IOBs is enqueued to the pipes for processing.

As each IOB is processed by the pipe and is retired, the HCD must update the status and length fields in the IOB and call usbdPostIOB. If the pipe halts due to an error, the pipe must remain halted and all pending IOBs must remain pending. It is the responsibility of the USBD to stop pending IOBs as required and to reactivate the pipe.

### Execution Environment

This function can be called from the interrupt (HCD_OFFLEVEL priority or lower) or process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdPipeStatus*

### Purpose

Obtains the pipe status.

### Syntax

```
#include <hcdi.h>
USBstatus hcdPipeStatus(pHCDI, hpipe);
PHCDI pHCDI;
USBhandle hpipe;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **hpipe** | Pipe handles returned by the *hcdPipeConnect* function. |

### Description

This function is an HCD supplied function that the USBD calls to obtain the status of the pipe from the host perspective. The function returns USBD_ACTIVE if the pipe is active and ready to receive I/O requests. The function returns USBD_HALTED if the pipe is halted.

### Execution Environment

This function can be called from the interrupt (HCD_OFFLEVEL priority or lower) or process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

### *hcdPipeAbort*

## Purpose

Aborts previously initiated device I/O operations.

## Syntax

```
#include <hcdi.h>
USBstatus hcdPipeAbort(pHCDI, pIOB)
PHCDI pHCDI;
PIOB pIOB;
```

## Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **pIOB** | Pointer to first IOB of the transaction to be aborted. |

## Description

This function is an HCD supplied function that the USBD calls to cancel the processing of an IOB. The pipe specified by the IOB is already halted before the *hcdPipeAbort* function is called. The HCD must remove only the specified IOB from the pipe, set the status field in the IOB to USBD_ABORTED, set the length field in the IOB to zero, and call the *usbdPostIOB* function. If the specified IOB has already processed by the pipe, the HCD must exit with a return code of USBD_SUCCESS because this is not an error.

## Execution Environment

This function can be called from the interrupt (HCD_OFFLEVEL priority or lower) or process environment. This function can also be called from the **usbdPostIOB** routine.

## Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

### *hcdPipeHalt*

## Purpose

Halts the specified pipe.

## Syntax

```
#include <hcdi.h>
USBstatus hcdPipeHalt(pHCDI, hpipe)
PHCDI pHCDI;
USBhandle hpipe;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **hpipe** | Pipe handles returned by the *hcdPipeConnect* function. |

### Description

This function is an HCD supplied function, called by the USBD to halt a pipe from the perspective of the host controller. All pending I/O operations remain pending. This function returns after the pipe is halted. If the specified pipe is halted when *hcdPipeHalt* function is called, the HCD must exit immediately with a return code of USBD_SUCCESS bcause this is not an error.

### Execution Environment

This function can only b called from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdPipeClear*

### Purpose

Clears halt condition and resume operation of a pipe.

### Syntax

```
#include <hcdi.h>
USBstatus hcdPipeClear(pHCDI, hpipe)
PHCDI pHCDI;
USBhandle hpipe;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **hpipe** | Pipe handles returned by the *hcdPipeConnect* function. |

### Description

This function is an HCD supplied function, called by the USBD to resume a pipe from the perspective of the host. This command does not affect the status of the pipe from the device perspective and it does not affect the value of the data synchronization toggle bit. If the specified pipe is active when the *hcdPipeClear* function is called, the HCD must set the return code to USBD_SUCCESS because this is not an error.

### Execution Environment

This function can be called from the interrupt (HCD_OFFLEVEL priority or lower) or process environment. This function can also be called from the **usbdPostIOB** routine.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdPipeResetToggle*

### Purpose

Resets data toggle flag to DATA0.

### Syntax

```
#include <hcdi.h>
USBstatus hcdPipeResetToggle(pHCDI, hpipe)
PHCDI pHCDI;
USBhandle hpipe;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **hpipe** | Pipe handles returned by the hcdPipeConnect function. |

### Description

This function is an HCD supplied function, called by the USBD to reset the data synchronization toggle bit to DATA0. The specified pipe handle specifies a halted bulk or interrupt pipe.

### Execution Environment

This function can be called from the interrupt (HCD_OFFLEVEL priority or lower) or process environment. This function can also be called from the **usbdPostIOB** routine.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdPipeAddIOB*

### Purpose

Assigns an extra buffer resources to an already connected pipe.

### Syntax

```
#include <hcdi.h>
USBstatus hcdPIpeAddIOB(pHCDI, hpipe,number)
PHCDI pHCDI;
USBhandle hpipe;
int number;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **hpipe** | Pipe handles returned by *hcdPipeConnect* function. |
| **number** | Number of extra buffers that are added. |

### Description

This function is an HCD supplied function, which is called by the USBD when increases the maximum number of outstanding I/O buffers in addition to what was specified through the **PIPEDESCRIPTOR** structure when the USBD is connected to the pipe. The HCD must use this call to allocate any additional resources that are required to handle the new allocation.

### Execution Environment

This function can be called from the process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdShutdownComplete*

### Purpose

Indicates that USBD has completed its HC shutdown operation.

### Syntax

```
#include <hcdi.h>
USBstatus hcdShutdownComplete(pHCDI)
PHCDI pHCDI;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |

### Description

This function is an HCD supplied function, called by the USBD to inform the HCD that the usbdReqHCshutdown request is completed. The HCD can now reset the host controller.

### Execution Environment

This function can only b called from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdGetFrame*

### Purpose

Gets the frame number.

### Syntax

```
#include <hcdi.h>
USBstatus hcdGetFrame (pHCDI, uint8_t speed, uint16_t *pFrame);
PHCDI pHCDI;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to HCDI structure. |
| **speed** | Current speed. |
| **pFrame** | Location to save the current frame number. |

### Description

This function is an HCD supplied function, called by the USBD to get the current frame number.

### Execution Environment

This function can only be called from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdDevAlloc*

### Purpose

Requests to allocate the xHCI adapter driver resources for the newly detected USB device.

### Syntax

```
#include <hcdi.h>
USBstatus hcdDevAlloc(pHCDI, devhdl )
PHCDI pHCDI;
USBhandle *devhdl
```

## Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to HCDI structure. |
| **devhdl** | Pointer to the location the USB logical device handle is placed. Maps to the **_DEVINFO** structure defined in **/usr/include/sys/hcdi.h** file. |

## Description

This call vector is specific only for the xHCI host controllers, that support the USB 3.0 protocol. This call vector is started by the USBD when it detects the attachment of a USB logical device. The call is started to inform the xHCI adapter driver about the presence of a new device and to request it to allocate and initialize its internal resources to configure and communicate with this device. Upon invocation, the HCD sends an **ENABLE SLOT** command to the xHCI to allocate a slot ID for this device and wait for its completion. If the command is successful, it populates an entry corresponding to the slot ID in the Device Context Base Address Array. It initializes the default control endpoint of this device by issuing the **Address Device** command to the xHCI to allocate a device address for this device and wait for its completion. Upon successful completion, it copies the device address for this device into the **_DEVINFO** data structure.

## Execution Environment

This function can only be called from within the process environment.

## Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

### *hcdDevFree*

## Purpose

Requests to free adapter driver and xHC resources for a current device.

## Syntax

```
#include <hcdi.h>
USBstatus hcdDevFree(pHCDI, devhdl )
PHCDI pHCDI;
USBhandle *devhdl
```

## Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **devhdl** | Pointer to the location the USB logical device handle returned by call to the *hcdDevAlloc* function. |

## Description

This call vector is specific only for xHCI host controllers, that support the USB 3.0 protocol. This is a HCD supplied function, called by the USBD when a device removal is detected by the USBD. The HCD requests

the xHC (host controller hardware) to free resources for this detached device by issuing a **Disable Slot** command. After the command is successful, the HCD frees the adapter-driver level logical device structure and the memory that is allocated for the **Device Context** data structures.

### Execution Environment

This function can only be called from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *hcdConfigPipes*

### Purpose

Requests to allocate and configure the device endpoint-related resources.

### Syntax

```
#include <hcdi.h>
USBstatus hcdConfigPipes(phcdi, devhdl, pDesc )
PHCDI pHCDI
USBhandle devhdl
PDESCIDX pDesc
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **devhdl** | Pointer to the location the USB logical device handle returned by call to the *hcdDevAlloc* function. |
| **pDesc** | Pointer to device descriptor list. |

### Description

This call vector is specific only for the xHCI host controllers, that support USB 3.0 protocol. It is started by USBD during the enumeration of USB logical device. As part of enumeration the USB system driver reads in the descriptors of the new device to configure the device for its intended use. Upon invocation, this function allocates and initializes a Transfer Ring for each endpoint except the default control endpoint. It then initializes the endpoint context data structure that is associated with each endpoint and issues a **Configure Endpoint** command to the xHC to inform about the endpoints that need to be added and then, waits for the successful completion.

### Execution Environment

This function can only be called from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |

| Value | Description |
| --- | --- |
| All others | Failure |

### *hcdUnconfigPipes*

## Purpose

Requests to allocate and configure the device endpoint-related resources.

## Syntax

```
#include <hcdi.h>
USBstatus hcdUnConfigPipes(hcdip, devhdl )
PHCDI hcdip
USBhandle devhdl
```

## Parameter

| Item | Description |
| --- | --- |
| **pHCDI** | Pointer to the HCDI structure. |
| **devhdl** | Pointer to the location the USB logical device handle is placed. |

## Description

This call vector is specific only for the xHCI host controllers, that support USB 3.0 protocol. This call vector is started by the USBD system driver when it detects that a device is removed from the system. It informs the HC to free the resources that are associated with the device and its currently configured endpoints. Upon invocation, this function issues **`Configure Endpoint`** command with Deconfigure flag set. It frees the rings that are allocated for all endpoints except the default control endpoint.

## Execution Environment

This function can only be called from within the process environment.

## Return Values

| Value | Description |
| --- | --- |
| USBD_SUCCESS | Success |
| All others | Failure |

### *usbdBusMap*

## Purpose

Maps pinned kernel memory so that it can be bus mastered by host controller.

## Syntax

```
#include <hcdi.h>
USBstatus usbdBusMap(pHCDI, addr, size, ppbusmap)
PHCDI pHCDI;
caddr_t addr;
int size;
PBUSMAP *ppbusmap;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **addr** | Pointer to memory block aligned on a word boundary that is allocated from kernel's pinned heap. The block must not cross a page boundary. |
| **size** | Size of memory block in bytes (must be less than or equal to 4096 bytes). |
| **ppbusmap** | Location to return address of the **BUSMAP** structure. |

### Description

This call vector is a USBD supplied function, called by the HCD to map pinned kernel memory so that it can be read from or written to the host controller.

### Execution Environment

This function can only b called from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *usbdunBusMap*

### Purpose

Unmaps the kernel memory.

### Syntax

```
#include <hcdi.h>
USBstatus usbdUnBusMap(pHCDI, pbusmap)
PHCDI pHCDI;
PBUSMAP pbusmap;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **PBUSMAP** | Pointer to BUSMAP structure returned by *usbdBusMap* function. |

### Description

This function is a USBD supplied function, called by the HCD to unmap previously mapped memory. All I/O referencing the memory block must is completed before this function is called.

### Execution Environment

This function can only be called from within the process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *usbdReqHCunregister*

### Purpose

Requests that the USBD close the communication channel to the host controller.

### Syntax

```
#include <hcdi.h>
USBstatus usbdReqHCunregister(pHCDI)
PHCDI pHCDI;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |

### Description

This function is a USBD supplied function that is called by the HCD when a host controller is unconfigured. This function causes the USBD to remove the host controller from its topology map. Any outstanding I/O is stopped and the appropriate client drivers are notified that the logical USB devices, which are connected to the controller, do not exists. A *hcdUnregisterHC* call is then made to the HCD and upon return, all system resources that are associated with the host controller are freed and control is returned to the HCD.

### Execution Environment

This function can only be called from the process environment only.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *usbdPostIOB*

### Purpose

Posts transaction complete.

### Syntax

```
#include <hcdi.h>
USBstatus usbdPostIOB(pHCDI, piob)
PHCDI pHCDI;
PIOB piob;
```

## Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |
| **piob** | Pointer to IOB associated with transaction. |

## Description

This call vector is started by the adapter driver to retire an IOB. Upon invocation, the driver unmaps the memory area that is associated with the IOB. If the IOB completes successfully and is the last IOB of an IRP transaction, it processes the IRP associated with the IOB and completes the IRP to the caller. If the IOB is completed with an unsuccessful status and is the last IOB of a transaction, the IRP that is associated with the IOB is completed and an unsuccessful status is returned to the client driver and all the IOBs outstanding on the pipe are started for bulk and interrupt endpoints. For control and isochronous endpoints the pipes are put in an unhalted state. If the IOB is completed with an unsuccessful state and is not the last IOB of a transaction, all the remaining IOBs of that transaction are started. It then checks whether the pipe is in halted state. If so, it sets the completion status of IOB to USBD_HALTED. This call can be made from process or interrupt environment.

## Execution Environment

This function can be called from the interrupt (HCD_OFFLEVEL priority or lower) or process environment.

## Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

### *usbdReqHCshutdown*

## Purpose

Requests USBD to shutdown all devices attached to HC.

## Syntax

```
#include <hcdi.h>
USBstatus usbdReqHCshutdown(pHCDI)
PHCDI pHCDI;
```

## Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |

## Description

This function is a USBD supplied function called by the HCD when a catastrophic event occurred and the HC is reset. The USBD aborts all pending I/O and delete all enumerated devices that are associated with the specified host controller.

This call is called when an error is detected with the adapter and the recovery of adapter driver from this error requires shutting down the adapter. Upon invocation, the function queues in a **shutdown host controller** command to the config proc of USBD. When the config proc is scheduled by the operating

system, the USB system driver walks through the device tree and for each device present in the device tree, ends all the outstanding I/O, disconnects all the pipes, and unmaps the mapped buffers that are associated with the client.

### Execution Environment

This function can be called from the interrupt (HCD_OFFLEVEL priority or lower) or process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## *usbdReqHCrestart*

### Purpose

Requests HC brings up and device enumeration.

### Syntax

```
#include <hcdi.h>
USBstatus usbdReqHCrestart(pHCDI)
PHCDI pHCDI;
```

### Parameter

| Item | Description |
|---|---|
| **pHCDI** | Pointer to the HCDI structure. |

### Description

This function is a USBD supplied function called by the HCD when the HC is successfully reset and reinitialize.

This call is started when an error is detected with the adapter and the recovery of adapter driver from this error requires restarting the adapter. Upon invocation, the function queues in a **restart host controller** command to the config proc of USBD. When the config proc is scheduled by the operating system, the USB system driver walks through the device tree and for each device present in the device tree, ends all the outstanding I/O, disconnects all the pipes, and unmaps the mapped buffers that are associated with the client. It then attaches the root hub to the host controller tree and kicks off the device enumeration and configuration process again

### Execution Environment

This function can be called from the interrupt (HCD_OFFLEVEL priority or lower) or process environment.

### Return Values

| Value | Description |
|---|---|
| USBD_SUCCESS | Success |
| All others | Failure |

## USB Error Recovery

The USB protocol allows the host controller to try control, bulk, and interrupt data transfers up to three times when a transmission error occurs. There is no error tries for isochronous data transfers.

Some errors and conditions that are detected by the USB device causes bulk and interrupt pipes to halt from the device perspective. The USB client driver is responsible for clearing the halted condition by sending a USB_CLEAR_FEATURE control message to the device requesting that the endpoint's halted condition to be cleared. This also resets the data toggle bit for the endpoint to DATA0 on both the host controller and the device reestablishing data toggle synchronization. After the control message is successfully processed by the device, the USB client driver must call the *usbdPipeClear* function to reactivates the pipe on the host side.

If a STALL handshake is received from the pipe or if all attempts are exhausted, bulk and interrupt pipes are halted from the host perspective and the failing IRP also all pending IRPs for the halted endpoint is tired. The IRP that caused the error has a status of USBD_STALL or USBD_ERROR; all other IRPs have a status of USBD_HALTED and are not sent to the device. Any IRP sent to a halted pipe is immediately be retired with a status of USBD_HALTED without being sent to the device. Typically a STALL handshake from a bulk or interrupt pipe means that the endpoint from the device perspective is halted. The USB client driver might clear the halt state from both the device and host perspective to resend IRPs as required. For control pipes, if a STALL handshake is received or if all attempts are exhausted then the failing IRP is retired but the pipe is not halted; IRPs continue to be processed in the order that they were received. Typically a STALL handshake from a control pipe means that the client driver has sent the unsupported command to endpoint. The USB client driver is responsible for properly handling this situation.

## A Typical USB Driver Transaction Sequence

A simplified sequence of events for a transaction through the USB driver stack is as follows.

- The USB client or class driver calls the USBDI *usbdPipeIO* call vector with the IRP data structure as a parameter.
- Each IRP is broken down into several page-sized IOBs by the USBD except the first and last IOBs whose transfer size might be less than page size. The IOBs are then built based on the pipe/endpoint transfer type that is associated with the IRP.
- The IOBs are then chained together and are passed to adapter device driver by starting a Host controller driver call vector *hcdPipeIO*. The host controller driver is responsible for processing each IOB and building the corresponding command element that is accepted by the HC architecture. The host controller driver also handles the IO completion.
- The adapter driver off level I/O completion handler then removes these completed IOBs and updates the status of each IOB with the completion status of the USB transfer.
- It then calls the *usbdPostIOB* call vector to pass these completed IOBs to the USBD layer.
- When the last IOB associated with a current IRP is completed, the USBD posts the status of the completed IRP to the USB Client Driver by calling the completion callback that is associated with the IRP.

## USB Driver Internal Commands

During the initialization of the system, the USB bus driver issues several commands to discover the topology of each host controller.

These commands are internal to the USBD and are not associated with any client or operating system requests. To issue the requests, the USBD must generate and map the IRPs and the associated data buffers. In addition to topology discovery, the USB Protocol driver must configure the USB devices that are discovered for correct operation. The USB Protocol driver configures the USB devices by reading the descriptors that are associated with the devices and then selecting a configuration on the USB device. After configuration is complete, these devices are ready for operation and can accept the client or operating system commands.

In addition to the boot time, topology, discovery, and device configuration, the USB bus driver must support run time device attachments and removals and configure the devices.

# The IRP Structure

When the client driver wants performs I/O operations to a USB device, it calls the *usbdPipeIO* call vector.

The IRP structure is passed as one of the parameters of the call vector and is used for communication between the client driver and the USBD driver.

## IRP Structure

The IRP structure contains certain fields that are used to pass I/O commands and associated parameters to the USBD device driver. Other fields within this structure are used to pass returned status back to the client driver.

The IRP structure is defined in the **/usr/include/sys/usbdi.h** file.

Fields in the IRP structure are used as follows:

- The **pNext** field is used to group together numerous IRPs from the client driver and it is pointing to next IRP in the chain.
- The **hPipe** field refers to the pipe handle of the USBD internal data structure. The USBD uses the **hPipe** field of the IRP to identify the corresponding pipe that is associated with the endpoint of current USB device.
- The **pBuffer** field refers to the virtual address of the buffer.
- The **IOCompleteCallBack** routine refers to the callback that is called upon completion of the IRP from the bus driver standpoint.
- The **clientPriv** data is passed as a parameter to the **IOCompleteCallBack** routine upon completion of the processing of the corresponding IRP. This pointer is opaque to the bus driver and the USBD will not use it.
- The **transSize** field identifies the length of the transaction in bytes if there is a data transfer associated with the IRP.
- The **flags** field identifies whether the short transfer from device is successful and if the buffer associated with the **pBuffer** field is already mapped.
- The **status** field identifies the completion status that is associated with the IRP.
- The **frame** field identifies the frame number that is associated with the isochronous transfer. It identifies the frame number for low and full speed endpoints and the micro frame number for high speed and super speed isochronous endpoints.

# The IOB Structure

When the USBD driver performs an IO to a USB device through the host controller it starts the *hcdPipeIO* call vector.

The IOB structure is passed as one of the parameters of the call vector and is used for communication between the USBD driver and the adapter driver.

## IOB Structure concepts

The IOB structure contains certain fields that are used to pass associated parameters to the adapter device driver. Other fields within this structure are used to pass returned status back to the USBD driver. The IOB structure is defined in the **/usr/include/sys/usbdi.h** file.

- The **pNext** field is used to group together the IOBs
- The **hPipe** field is used by the xHCD to identify the logical hardware pipe that is associated with the endpoint of a current USB device.
- The **pBuffer** field points to the virtual address of the buffer.

- The **busAddr** field points to the bus address associated with the **pBuffer** field.
- The **length** field points to the length of the transaction in bytes if data transfer is associated with the IOB.
- The **flags** field identifies whether the short transfer from device is successful. It also identifies the type of the packet.
- The **status** field indicates the completion status of the IOB.
- The **frame** field indicates the frame number that is associated with the I/O request. This field is only valid for isochronous endpoints.

# USB Adapter Driver ioctl command

The following ioctl command is supported by the USB adapter device driver:

## HCD_REGISTER_HC

### Purpose

Registers the USB host Controller with the USB system driver.

### Syntax

```
int ioctl (file, USBD_REGISTER_SINGLE_HC, arg)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that are obtained when the USBD special file was opened. |
| **arg** | Pointer to the integer that contains 32-bit devno of the USB host controller. |

### Description

If successful, this ioctl registers the specified host controller with the USBD and allows clients to talk to devices connected to the controller. There is no specific IOCTL to unregister a hardware controller. It stays registered until either the USBD is unconfigured or the host controller is unconfigured. The host controller driver requests the USBD to unregister the host controller through a call vector (See the usbdReqHCunregister). For additional information, please see Host Controller Registration.

### Execution Environment

This function can be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **EFAULT** | Indicates either incorrect size of the call vector or incorrect version of the call vector data structure. |
| **EBUSY** | Indicates that adapter hardware is inaccessible. |
| **EINVAL** | Indicates that the Host controller is already registered with USBD. |

# USB Protocol and Bus Driver ioctl commands

The following ioctl commands are supported by the USB protocol or bus driver:

## USBD_REGISTER_SINGLE_HC

### Purpose

Register single USB Host Controller with the USB system driver.

### Syntax

```
int ioctl (file, USBD_REGISTER_SINGLE_HC, arg)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that are obtained when the USBD special file was opened. |
| **arg** | Pointer to the integer that contains 32-bit devno of the USB host controller. |

### Description

If successful, this ioctl registers the specified host controller with the USBD and allows clients to talk to devices connected to the controller. There is no specific IOCTL to unregister a hardware controller. It stays registered until either the USBD is unconfigured or the host controller is unconfigured. In the latter case, the host controller driver asks the USBD to unregister the host controller through a call vector (See the usbdReqHCunregister).

### Execution Environment

This function can be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **-1** | Failure (check errno for specific failure) |

## USBD_REGISTER_MULTI_HC

### Purpose

Register USB Host Controller with USB system driver.

### Syntax

```
int ioctl (file, USBD_REGISTER_MULTI_HC, arg)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that are obtained when the USBD special file was opened. |
| **arg** | Pointer to USB host controller information structure. |

### Description

If successful, this ioctl registers the all USB host controller listed in the **usb_adapterhc_info** structure with the USBD and allow clients to talk to devices connected to the controller. There is no specific IOCTL to unregister a hardware controller. It stays registered until either the USBD is unconfigured or the host controller is unconfigured. In the latter case, the host controller driver asks the USBD to unregister the host controller through a call vector **usbdReqHCunregister**.

This ioctl should be invoked only by the *cfgusb config* method during enumeration and individual USB adapter configuration methods should use USBD_REGISTER_SINGLE_HC for registering single host controller instance.

### Execution Environment

This function might be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **-1** | Failure (check errno for specific failure) |

## USBD_ENUMERATE_DEVICE

### Purpose

Gets a list of the USB logical devices (excluding hubs) connected to a host controller.

### Syntax

```
int ioctl (file, USBD_ENUMERATE_DEVICE, arg)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that is obtained when the USBD special file was opened. |
| **arg** | Address of the USBENUM structure that is aligned on a 4-byte boundary (see the USBENUM structure in the **/usr/include/sys/usbdi.h** file). |

### Description

This ioctl returns a description of each logical USB device that is connected to the specified host controllers sans any hubs. The description is returned in the form of the `usb_device_t` structure. The array of returned structures is encapsulated within a USBENUM structure whose length is specified by the caller.

When this function is invoked, the **devno** and **buffSize** fields within the USBENUM structure must be initialized. The **devno** field must contain the 32-bit devno of the host controller to be enumerated while

the**buffSize** field must indicate the number of bytes that are available to buffer the returned array of `usb_device_t` structures. If the area is small, the number of returned structures is truncated to fit the available space. The caller can detect this condition by noting that the number of returned the `usb_device_t` structures is less than the number of discovered logical devices.

This operation returns a list of all logical devices that are connected behind a current host controller excluding any hubs. If the passed in buffer size is too small to contain the entire list, the list is truncated to fit the allocated buffer size. The *numDevDisc* parameter in the list specifies the total number of devices that are detected behind the host controller and the *numDevEnum* parameter specifies the total number of devices that are enumerated. This value can be smaller than *numDevDisc* parameter if the allocated buffer size is small to fit in the entire list.

### Execution Environment

This function might be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **ENODEV** | Indicates that there is no host controller that is associated with the passed in device number. |
| **EFAULT** | Indicates that the user has insufficient authority to access the data. |
| **EIO** | Indicates that a permanent I/O error occurred while referencing data. |

## USBD_ENUMERATE_ALL

### Purpose

Gets a list of the USB logical devices connected to a host controller.

### Syntax

```
int ioctl (file, USBD_ENUMERATE_ALL, arg)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that is obtained when the USBD special file was opened. |
| **arg** | Address of the USBENUM structure that is aligned on a 4-byte boundary (see the USBENUM structure in the **/usr/include/sys/usbdi.h** file). |

### Description

This ioctl behaves the same as the USBD_ENUMERATE_DEVICE ioctl except that it also includes all hubs other than the root hub.

### Execution Environment

This function can be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **ENODEV** | Indicates that there is no host controller that is associated with the passed in device number. |
| **EFAULT** | Indicates that the user has insufficient authority to access the data. |
| **EIO** | Indicates that a permanent I/O error occurred while referencing data |

## USBD_ENUMERATE_CFG

### Purpose

Gets a list of the USB logical devices connected to a host controller.

**Warning:** This IOCTL must be used only by the configuration method of the USB System Device Driver.

### Syntax

```
int ioctl (file, USBD_ENUMERATE_CFG, arg)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that is obtained when the USBD special file was opened. |
| **arg** | Address of the **USBENUMCFG** structure that is aligned on a 4-byte boundary (see USBENUMCFG structure in the **/usr/include/sys/usbdi.h** file). |

### Description

This ioctl behaves the same as the USBD_ENUMERATE_DEVICE ioctl except that it also returns client device selection information. The selection information uniquely identifies device or client pairing and allows the configuration method to correlate enumerated devices with their ODM instances.

### Execution Environment

This function may be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **ENODEV** | Indicates that there is no host controller that is associated with the passed in device number. |
| **EFAULT** | Indicates that the user has insufficient authority to access the data. |
| **EIO** | Indicates that a permanent I/O error occurred while referencing data. |

## USBD_GET_DESCRIPTORS

### Purpose

Gets standard the USB descriptors for a logical device.

### Syntax

```
int ioctl (file, USBD_GET_DESCRIPTORS, arg)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that is obtained when the USBD special file was opened. |
| **arg** | Address of the **USBDGD** structure that is aligned on a 4-byte boundary (see USBDGD structure that is defined in the **/usr/include/sys/usbdi.h** file). |
| **ext** | Not used and should be set to zero. |

### Description

Upon successful return from the ioctl, a DESCIDX structure is placed at the start of the specified buffer followed by the standard device descriptor, configuration descriptor, interface descriptor, endpoint descriptors, HID descriptor (if HID device), hub descriptor (if hub device), and string descriptors of the specified logical USB device. The DESCIDX structure provides direct addressability to the individual descriptors. String descriptors are reformed to NULL terminated ASCII strings for ease of use; all other descriptors adhere to the standard USB format. Because the size of the returned data is typically unknown, it is recommended that the ioctl be called twice. The first-time set **bufferLength** field equal to zero and *buffer* field to NULL. The ioctl fails with errno of ENOSPC; however **minBuffLength** field is returned indicating the required size of the buffer. The caller can then allocate the buffer and call the ioctl a second time with **bufferLength** field set to the correct value.

### Execution Environment

This function can be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **ENODEV** | Indicates that there is no host controller that is associated with the passed in device number. |
| **ENOSPC** | Indicates that the allocated buffer length is too small to fit the descriptor data. |

## USBD_CFG_CLIENT_UPDATE

### Purpose

Updates client connection information.

**Warning:** This IOCTL is for use ONLY by the configuration method of the USB System Device Driver.

### Syntax

```
int ioctl (file, USBD_CFG_CLIENT, arg)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that is obtained when USBD special file was opened. |
| **arg** | Address of the USBENUMCFG structure that is aligned on a 4-byte boundary (see the CLIENTUPDATE structure in the **/usr/include/sys/usbdi.h** file). |

### Description

This ioctl is used by the configuration procedure defined by the USBD to update the device selection criteria that are used by the client driver. Specifically, it updates the **hcdevno**, **addr**, **cfg**, and **intfc** fields to reflect the current values for the device that is managed by the client.

### Execution Environment

This function can be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **EINVAL** | Indicates wrong device selection criteria. |
| **ENODEV** | Indicates that no device is found with the device selection criteria. |

## USBD_OPEN_DEVICE

### Purpose

Opens a specific USB logical device.

### Syntax

```
int fp_ioctl (file, USBD_OPEN_DEVICE, arg, ext)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that is obtained when the USBD special file was opened. |
| **arg** | Address of an initialized DEVOPEN structure (See the DEVOPEN structure that is defined in the **/usr/include/sys/usbdi.h** file). |
| **ext** | Not used and must be set to zero. |

### Description

The client driver uses this fp_ioctl to establish a connection to a specific USB logical device as identified by information within the DEVSELECTOR structure. A USB logical device can be opened by only one client

driver at a time. After a client is opened the device, it must connect to a pipe before data flows to or from the device. This includes the default control pipe. The client driver must close any device that it is opened when it is not managing the device by calling *usbdCloseDevice* call vector.

Typically, a client driver must open the USB System Driver, issue a USBD_OPEN_DEVICE ioctl to open a specific USB device, and then close the USB System Driver. The client must then communicate with the USB system driver by using the handle returned by the USBD_OPEN_DEVICE ioctl and the interface macros located within the **usbdi.h**file.

To properly track the USB device must it be moved or replaced, it is strongly recommended that the client open the device when the client is configured and closed the USB device when the client is un-configured.

### Execution Environment

This function may be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **EINVAL** | Invalid parameter. |
| **ENODEV** | Indicates that the device with matching selection criteria does not exist. The selection criteria include matching class, subclass, and protocol. |

## USBD_OPEN_DEVICE_EXT

### Purpose

Opens a specific USB logical device.

### Syntax

```
int fp_ioctl (file, USBD_OPEN_DEVICE_EXT, arg, ext)
```

### Parameter

| Item | Description |
|---|---|
| **file** | File descriptor that is obtained when th USBD special file was opened. |
| **arg** | Address of an initialized DEVOPEN structure (See the DEVOPEN structure in the **/usr/include/sys/usbdi.h** file). |
| **ext** | Not used and must be set to zero. |

### Description

The client driver uses this fp_ioctl to establish a connection to a USB logical device as identified by information within the DEVSELECTOR structure. The ioctl is similar to the USBD_OPEN_DEVICE ioctl except that a client handle is allocated even when a USB logical device that matches the criteria specified through the DEVSELECTOR structure is not available. The USBD returns EAGAIN to indicate this condition. When EAGAIN is returned, the client driver must treat the device as being disconnected and wait for a reconnection call-back before preceding with device initialization.

### Execution Environment

This function can be called from the user process environment only.

### Return Values

| Value | Description |
|---|---|
| **0** | Indicates successful completion. |
| **EINVAL** | Invalid parameter. |
| **EAGAIN** | Returned if specified device does not exist; client driver must treat device as open and disconnected |

## Debug Facilities

You can use the available procedures for debugging a device driver that is under development.

Review the following command and kernel service descriptions before you debug a device driver:

- **dumpctrl** command
- **errinstall** command
- **errlogger** command
- **errmsg** command
- **errupdate** command
- **extendlv** command
- **livedumpstart** command
- **sysdumpdev** command
- **sysdumpstart** command
- **trace** command
- **trcrpt** command
- **errsave** kernel service
- **ras_register** kernel service
- **ras_control** kernel service
- **ldmp_setupparms** kernel service
- **livedump** kernel service
- **dmp_compspec** kernel service

**Related information**

Software Product Packaging

Changing or Removing a Paging Space

## System Dump Facility

Your system generates a system dump when a severe error occurs. System dumps can also be user-initiated by users with root user authority. A system dump creates a picture of your system's memory contents. System administrators and programmers can generate a dump and analyze its contents when debugging new applications.

System dumps can be assisted by firmware. Different from traditional system dumps that are generated before the partition is reinitialized, firmware-assisted system dumps take place when the partition is restarting. Firmware-assisted system dumps can be one of these types:

**Selective memory dump**
Selective memory dumps are triggered by or use AIX instances that must be dumped.

**Full memory dump**
The whole partition memory is dumped without any interaction with an AIX instance that is failing.

By default, the system generates a traditional system dump. After you explicitly enable a firmware-assisted system dump, it becomes the preferred system dump. However, if the configuration of the firmware-assisted system dump fails, the system generates a traditional system dump. A firmware-assisted system dump takes place under the following conditions:

- The platform supports firmware-assisted system dumps. AIX retrieves the property of firmware-assisted system dumps in the device tree to get the information.
- The memory size at system startup is equal to or greater than 1.5GB.
- You have not configured a traditional system dump.

A firmware-assisted dump cannot copy dump tables. Because the data is written on the next restart of the system, the dump tables, which are used to refer to the data, cannot be preserved.

RAS infrastructure components can be system-dump aware, allowing granular control of the amount of data that is dumped in a system dump by infrastructure components. Components that are system-dump aware can be excluded from a system dump to reduce the dump size. You can use the **dumpctrl** command to obtain information about which infrastructure components are registered for a system dump.

Use the **ras_register** kernel service to make an infrastructure component dump-aware. Use the RASCD_SET_SDMP_ON command that is passed to the **ras_control** kernel service to make an infrastructure component system-dump aware. See "Callback Commands for System Dumps" on page 366.

If your system stops with an 888 number flashing in the operator panel display, the system has generated a dump and saved it to a dump device (the condition only occurs with traditional system dumps).

Some of the error log and dump commands are delivered in an optionally installable package called **bos.sysmgt.serv_aid**. System dump commands included in the **bos.sysmgt.serv_aid** include the **sysdumpstart** command. See the Software Service Aids Package for more information.

To generate a system dump see the following topics:

### EMC PowerPath support for traditional and firmware assisted dump

To complete a traditional assisted dump or a firmware assisted dump, you must have an installation of EMC PowerPath Version 5.7, or later, or you must have an installation of EMC PowerPath Version 5.5.0.2, or earlier. For example, if you have and installation of EMC PowerPath Version 5.6, you cannot complete a traditional assisted dump or a firmware assisted dump.

## Configuring a Dump Device

When an unexpected system halt occurs, the system dump facility automatically copies selected areas of kernel data to the primary dump device. These areas include kernel segment 0, as well as other areas registered in the Master Dump Table by kernel modules or kernel extensions.

An attempt is made to dump to a secondary dump device if it has been defined.

**Restriction:** You cannot use firmware-assisted system dumps for the secondary dump device.

If your system has less than 4 GB of memory, the default dump device is the **/dev/hd6** logical volume, which is a paging logical volume. In addition, if the configuration is for a firmware-assisted system dump, you cannot configure the **/dev/hd6** logical volume, or any paging space, as the dump device.

Beginning with AIX® 6.1 with the 6100-01 Technology Level, you can configure an iSCSI device as the dump device for a firmware-assisted system dump.

**Restriction:** You can only use an iSCSI-dump logical volume that resides on the iSCSI boot disk.

Beginning with AIX® 6.1 Technology Level 6100-04, you can perform remote dumps on thin servers. To perform a remote dump on a thin server, you must define the relative dump resource on the NIM master and allocate the related dump resource to the thin server. The dump resource appears as an iSCSI disk to the NIM client and can only be used to configure the primary dump device. You can only configure firmware-assisted system dumps on primary dump devices. See Defining a dump resource for more information.

If a dump occurs to paging space (traditional system dumps only), the system will automatically copy the dump when the system is rebooted. By default, the dump is copied to the **/var/adm/ras** directory in the root volume group. See the **sysdumpdev** command for details on how to control dump copying.

Beginning with AIX® 6.1, you can only use compressed dumps. See the **sysdumpdev**, `uncompress` command, and **dmpuncompress** commands for more details.

The **dumpcheck** facility notifies you if your dump device needs to be larger, or if the file system containing the copy directory is too small. This notification appears in the system error log. If you need to increase the size of your dump device, see "Increasing the Size of a Dump Device" on page 366.

For maximum effectiveness, dumpcheck should be run when the system is most heavily loaded. At such times, the system dump is most likely to be at its maximum size. Also, even with dumpcheck watching the dump size, it may still happen that the dump won't fit on the dump device or in the copy directory at the time it happens. This could occur if there is a peak in system load right at dump time.

### *Including Device Driver Data*

To have your device driver data areas included in a system dump, you must register the data areas in the master dump table.

Use the **dmp_ctl** kernel service to add an entry to the master dump table or to delete an entry.

The syntax is as follows:

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

int dmp_ctl(op, data)
int op;
struct dmpctl_data *data;
```

Use the **ras_register** kernel service and the **ras_control** kernel service to register a component for a system dump, as shown in the following example:

```
/* Register a component as dump aware */
rv = ras_register(&Rascb,(char*)Compname,(ras_block_t)0,RAS_TYPE_OTHER,
     "sample component",RASF_DUMP_AWARE,sample_callback,NULL);
if (rv) return(KERROR2ERRNO(rv));

/* Make the component system dump aware. */
rv = ras_control(Rascb, RASCD_SET_SDMP_ON, 0, 0);
if (rv) return(KERROR2ERRNO(rv));
```

## Starting a System Dump

A user-initiated dump is different from a dump initiated by an unexpected system halt because the user can designate which dump device to use. When the system halts unexpectedly, a system dump is initiated automatically to the primary dump device.

**Attention:** Do not start a system dump if the flashing 888 number shows in your operator panel display. This number indicates your system has already created a system dump and written the information to your primary dump device. If you start your own dump before copying the information in your dump device, your new dump will overwrite the existing information. For more information, see "Checking the Status of a System Dump" on page 362.

**Tips:**

- You can use the **Dump** option of the **Restart Partition** function to initiate an AIX® stand-alone system dump on POWER5™ logical partitions that are managed by a Hardware Management Console (HMC). For more information about initiating a system dump during the restart of logical partitions using HMC, go to the IBM® Systems Hardware Information Center available at http://publib.boulder.ibm.com/eserver/ and search for *restart AIX® logical partitions*.

- You can initiate a system dump remotely using a modem or terminal server after enabling the AIX® remote-restart facility using the **smitty rrbtty** fast path. However, the AIX® remote restart facility does not work for a system (integrated serial) port on a POWER5™ system. Instead, enable serial port snoop. For more information about how to enable serial port snoop, go to the IBM® Systems Hardware Information Center and search for *enable serial port snoop*.
- While a logical partition is dumping, dump progress indicators (0c0, 0c2, 0c9, and so on) will appear on the HMC and in the LCD display. For more information about the dump status codes, go to the IBM® Systems Hardware Information Center and search for *dump progress indicators*.

If you have access to the **sysdumpstart** command, you can start a system dump using one of these methods:

### *Using the Command Line*

Use the following steps to choose a dump device, specify the dump type, initiate the system dump, and determine the status of the system dump.

**Requirement:** You must have root user authority to start a dump using the **sysdumpstart** command.

1. Check which dump device is appropriate for your system (the primary or secondary device) by using the following **sysdumpdev** command:

   ```
   sysdumpdev -l
   ```

   This command lists the current dump devices. You can use the **sysdumpdev** command to change device assignments.
2. Start the system dump using the **sysdumpstart** command. The **sysdumpstart** command starts a system dump on the default primary dump device. Use the **-p** flag to start a system dump to the primary dump device. Use the **-s** flag to start a system dump to the secondary dump device. Use the **-t** flag to specify a traditional system dump (**-t traditional**). You cannot force a firmware-assisted system dump if the traditional system dump is configured.

   ```
   sysdumpstart -p
   ```

   **Restriction:** You cannot start a firmware-assisted system dumps on the secondary dump device.
3. If a code shows in the operator panel display, refer to “Checking the Status of a System Dump” on page 362. If the operator panel display is blank, the dump was not started. Try again using the Reset button.

### *Using SMIT*

Use the following SMIT commands to choose a dump device and start the system dump:

**Note:** You must have root user authority to start a dump using SMIT. SMIT uses the **sysdumpstart** command to start a system dump.

1. Check which dump device is appropriate for your system (the primary or secondary device) by using the following SMIT fast path command:

   ```
   smit dump
   ```
2. Choose the **Show Current Dump Devices** option and write the available devices on notepaper.
3. Enter the following SMIT fast path command again:

   ```
   smit dump
   ```
4. Choose either the primary (the first example option) or secondary (the second example option) dump device to hold your dump information:

   ```
   Start a Dump to the Primary Dump Device
   ```

OR

```
Start a Dump to the Secondary Dump Device
```

Base your decision on the list of devices you made in step 2.

5. Choose the type of dump you want, if you choose the primary dump device to hold your dump information.

   To start a system dump according to the current dump configuration, choose the following type:

   ```
   Start a System Dump to the Primary Dump Device
   ```

   To force a full memory dump, a selective memory dump, or a traditional system dump, choose the corresponding type:

   ```
   Start a Full Memory Dump to the Primary Dump Device
   Start a Selective Memory Dump to the Primary Dump Device
   Start a Traditional System Dump to the Primary Dump Device
   ```

6. Click **Reset** to start a dump again if the dump was not started (the operator panel display is blank). See "Checking the Status of a System Dump" on page 362 if a value shows in the operator panel display.

   To start a dump with the **Reset** button or a key sequence you must have the key switch, or mode switch, in the Service position, or have set the Always Allow System Dump value to true. To do this:

   a. Use the following SMIT fast path command:

      ```
      smit dump
      ```

   b. Set the Always Allow System Dump value to true. This is essential on systems that do not have a mode switch.

### *Using the Reset Button*

In AIX® 5.3 and subsequent releases, pressing the reset button always dumps to the primary dump device. This is also true for LPAR systems running AIX® 5.2.

Start a system dump with the Reset button by doing the following (this procedure works for all system configurations and will work in circumstances where other methods for starting a dump will not work):

1. If your machine has a key mode switch, do one of the following:
   - Turn the key mode switch to the Service position.
   - Set **Always Allow System Dump** to true.
   - Press the **Reset** button.
2. If your machine does not have a key mode switch, set **Always Allow System Dump** to true and press the **Reset** button.

Your system writes the dump information to the primary dump device.

**Note:** The procedure for using the reset button can vary, depending upon your hardware configuration.

### *Using Special Key Sequences*

This section tells how to start a system dump with special key sequences.

Start a system dump with special key sequences by doing the following:

1. Turn your machine's mode switch to the Service position, or set Always Allow System Dump to true.
2. Press the Ctrl-Alt 1 key sequence to write the dump information to the primary dump device, or press the Ctrl-Alt 2 key sequence to write the dump information to the secondary dump device.

You can start a system dump by this method *only* on the native keyboard.

### *Using the HMC Command Line*

This section explains how to start a system dump using the Hardware Management Console (HMC) command line interface.

1. At the HMC command prompt enter the command:

   ```
   startdump -m <managed-system> -t sys
   ```

   where <managed-system> is the name of the managed system on which you want to initiate the system dump.

   For example the command:

   ```
   startdump -m Server-7895-23X-SN2150037 -t sys
   ```

   initiates a system dump on the managed system Server-7895-23X-SN2150037.
2. After entering the **startdump** command, you will see the following confirmation:

   ```
   You have requested to start a system dump of the managed system.
   This will halt the managed system.  When the dump has completed,
   the managed system will be rebooted.  Do you want to continue (0 = no, 1 = yes)?
   ```

   Enter **1** to confirm that you want to initiate the system dump.

## Checking the Status of a System Dump

When a system dump is taking place, status and completion codes are displayed in the operator panel display on the operator panel. When the dump is complete, a `0cx` status code displays if the dump was user initiated, a flashing 888 displays if the dump was system initiated.

You can check whether the dump was successful, and if not, what caused the dump to fail. If a `0cx` is displayed, see "Status Codes" on page 362 below.

**Note:** If the dump fails and upon reboot you see an error log entry with the label `DSI_PROC` or `ISI_PROC`, and the Detailed Data area shows an **EXVAL** of `000 0005`, this is probably a paging space I/O error. If the paging space (probably**/dev/hd6**) is the dump device or on the same hard drive as the dump device, your dump may have failed because of a problem with that hard drive. You should run diagnostics against that disk.

## Status Codes

Find your status code in the following list, and follow the instructions:

| Item | Description |
|---|---|
| **000** | The kernel debugger is started. If there is an ASCII terminal attached to one of the native serial ports, enter `q dump` at the debugger prompt (>) on that terminal and then wait for flashing 888s to appear in the operator panel display. After the flashing 888 appears, go to "Checking the Status of a System Dump" on page 362. |
| **0c0** | The dump completed successfully. Go to "Copying a System Dump" on page 363. |
| **0c1** | An I/O error occurred during the dump. Go to "System Dump Facility" on page 357. |
| **0c2** | A user-requested dump is not finished. Wait at least 1 minute for the dump to complete and for the operator panel display value to change. If the operator panel display value changes, find the new value on this list. If the value does not change, then the dump did not complete because of an unexpected error. |
| **0c4** | The dump ran out of space . A partial dump was written to the dump device, but there is not enough space on the dump device to contain the entire dump. To prevent this problem from occurring again, you must increase the size of your dump media. Go to "Increase the Size of a Dump Device" on page 366. |
| **0c5** | The dump failed because of an internal error. |

| Item | Description |
|---|---|
| **0c8** | The dump device has been disabled. The current system configuration does not designate a device for the requested dump. Enter the **sysdumpdev** command to configure the dump device. |
| **0c9** | A dump started by the system did not complete. Wait at least 1 minute for the dump to complete and for the operator panel display value to change. If the operator panel display value changes, find the new value on the list. If the value does not change, then the dump did not complete because of an unexpected error. |
| **0ca** | A firmware-assisted system dump is not finished yet. System startup resumes after the dump completes. |
| **0cb** | A firmware-assisted selective memory dump is started. Wait at least 10 minutes for the dump to complete and for the operator-panel display value to change. If the operator-panel display value changes, find the new value on the list. If the value does not change, the dump did not complete because of an unexpected error. |
| **0cc** | An error occurred dumping to the primary device; the dump has switched over to the secondary device. Wait at least 1 minute for the dump to complete and for the three-digit display value to change. If the three-digit display value changes, find the new value on this list. If the value does not change, then the dump did not complete because of an unexpected error. |
| **c20** | The kernel debugger exited without a request for a system dump. Enter the **quit dump** subcommand. Read the new three-digit value from the LED display. |

## Copying a System Dump

Your dump device holds the information that a system dump generates, whether the information is generated by the system or a user. You can copy this information to tape and deliver the material to your service department for analysis.

**Note:** If you intend to use a tape to send a snap image to IBM for software support, the tape must be one of the following formats: **8 mm, 2.3 Gb** capacity, **8 mm, 5.0 Gb** capacity, or **4 mm, 4.0 Gb** capacity. Using other formats will prevent or delay software support from being able to examine the contents.

There are two procedures for copying a system dump, depending on whether you are using a dataless workstation or a non-dataless machine.

### *Copying a System Dump on a Dataless or Diskless Workstation*

On a dataless or diskless workstation, the dump is automatically copied to the server when the workstation is rebooted after the dump. The dump remains available to the workstation.

Copy a system dump on a dataless or diskless workstation by performing the following tasks:

1. Reboot in normal mode.
2. Locate the system dump.
3. Copy the system dump from the server.

*Reboot in Normal mode*
Use this procedure to reboot your machine in Normal mode.

To reboot in normal mode:

1. Turn off the power on your machine.
2. Turn the mode switch to the Normal position.
3. Turn on the power on your machine.

*Locate the System Dump*
Follow these steps to locate the system dump.

To locate the dump:

1. Log on to the server.

2. Use the **lsnim** command to find the dump object for the workstation. For this example, the workstation's object name on the server is `worker`.

   ```
   lsnim -l worker
   ```

   The dump object is displayed on the line:

   ```
   dump = dumpobject
   ```

3. Use the **lsnim** command again to determine the path of the object:

   ```
   lsnim -l dumpobject
   ```

   The path name displayed is the directory containing the dump. The name of the dump is the same as the object for the dataless or diskless workstation.

*Copy the System Dump from the Server*
You can copy the dump like any other file, or you can aggregate the dump to workstation data by using the **snap** command.

To copy the dump to tape, use the **tar** command:

```
tar -c
```

To copy to a tape other than the **/dev/rmt0** tape:

```
tar -cftapedevice
```

To copy the dump back from the external media (such as a tape drive), use the **tar** command. Enter the following to copy the dump from the **/dev/rmt0** tape:

```
tar -x
```

To copy the dump from any other media, enter:

```
tar -xftapedevice
```

To aggregate the dump to workstation data, run the **nim -o snap** command or select the snap operation in the **smit nim_mac_op** panel. Both methods remotely execute the **snap** command. The collected data is located in the same directory as the dump directory. You can save the collected data to a tape, external media, or any other media as previously described.

## *Copying a System Dump on a Non-Dataless or Non-Diskless Machine*

Copy a system dump on a non-dataless or a non-diskless machine by performing the following tasks:

1. Reboot your machine.
2. Copy the system dump.

*Reboot Your Machine*
Use this procedure to reboot your machine in Normal mode.

Reboot in Normal mode using the following steps.

1. Turn off the power on your machine.
2. Turn the mode switch to the Normal position.
3. Turn on the power on your machine.

If your system shows the login prompt, go to “Copy a System Dump after Rebooting in Normal Mode” on page 365.

If your system stops with a number in the operator panel display instead of showing the login prompt, reboot your machine from Maintenance mode, then go to "Copy a System Dump after Booting from Maintenance Mode" on page 365.

*Copy a System Dump after Rebooting in Normal Mode*
While copying a system dump on a non-dataless or non-diskless machine, if you reboot the machine in normal mode, you must perform the following steps.

After rebooting in Normal mode, copy a system dump by doing the following:

1. Log in to your system as root user.
2. Copy the system dump to tape using the following **snap** command, where the number sign (#) is the number of your available tape device (the most common is **/dev/rmt0**):

   ```
   /usr/sbin/snap -gfkD -o /dev/rmt#
   ```

   To find the correct number of the available tape device, enter the following **lsdev** command, and look for the tape device listed as `Available`:

   ```
   lsdev -C -c tape -H
   ```

   **Note:** If your dump went to a paging space logical volume, it has been copied to a directory in your root volume group, **/var/adm/ras**. For more information, see Configure a Dump Device and the **sysdumpdev** command. These dumps are still copied by the **snap** command. The **sysdumpdev -L** command lists the exact location of the dump.
3. To copy the dump back from the external media (such as a tape drive), use the **pax** command. Enter the following to copy the dump from **/dev/rmt0**:

   ```
   pax -rf/dev/rmt0
   ```

   To copy the dump from any other media, enter:

   ```
   tar -xftapedevice
   ```

*Copy a System Dump after Booting from Maintenance Mode*
While copying a system dump on a non-dataless or non-diskless machine, if you reboot the machine in maintenance mode, you must perform the following steps.

**Note:** Use this procedure only if you cannot boot your machine in Normal mode.

1. After booting from Maintenance mode, copy a system dump or tape using the following **snap** command:

   ```
   /usr/sbin/snap -gfkD -o /dev/rmt#
   ```
2. To copy the dump back from the external media (such as a tape drive), use the **tar** command. Enter the following to copy the dump from the **/dev/rmt0** tape:

   ```
   tar -x
   ```

   To copy the dump from any other media, enter:

   ```
   tar -xftapedevice
   ```

## Increase the Size of a Dump Device

Refer to the following to determine the appropriate size for your dump logical volume and to increase the size of either a logical volume or a paging space logical volume.

### *Determining the Size of a Dump Device*

The size required for a dump is not a constant value because the system does not dump paging space; only data that resides in real memory can be dumped. Paging space logical volumes will generally hold the system dump. However, because an incomplete dump may not be usable, follow the procedure below to make sure that you have enough dump space.

When a system dump occurs, all of the kernel segment that resides in real memory is dumped (the kernel segment is segment 0). Memory resident user data (such as u-blocks) are also dumped.

The minimum size for the dump space can best be determined using the **sysdumpdev -e** command. This gives an estimated dump size taking into account the memory currently in use by the system. If dumps are being compressed, then the estimate shown is for the compressed size of the dump, not the original size. In general, compressed dump size estimates will be much higher than the actual size. This occurs because of the unpredictability of the compression algorithm's efficiency. You should still ensure your dump device is large enough to hold the estimated size in order to avoid losing dump data.

For example, enter:

```
sysdumpdev -e
```

If **sysdumpdev -e** returns the message, `Estimated dump size in bytes: 9830400`, then the dump device should be at least 9830400 bytes or 12MB (if you are using three 4MB partitions for the disk).

### *Determining the Type of Logical Volume*

Paging space logical volumes will generally hold the system dump. However, because an incomplete dump may not be usable, follow the procedure below to make sure that you have enough dump space.

1. Enter the **sysdumpdev** command to list the dump devices. The logical volume of the primary dump device might be **/dev/lg_dumplv** or **/dev/hd7**.

   **Note:** You can also determine the dump devices using SMIT. Select the **Show Current Dump Devices** option from the System Dump SMIT menu.
2. Determine your logical volume type by using SMIT. Enter the SMIT fast path **smit lvm** or **smitty lvm**. You will go directly to Logical Volumes. Select the **List all Logical Volumes by Volume Group** option.

   Find your dump volume in the list and note its Type (in the second column). For example, the logical volume type might be sysdump if the dump device is **lg_dumplv**, or paging if the dump device is **hd6**.

### *Increasing the Size of a Dump Device*

Increasing the size of dump device depends on whether your dump device is a paging space or sysdump.

If you have confirmed that your dump device is a paging space, refer to Changing or Removing a Paging Space in *Operating system and device management* for more information.

If you have confirmed that your dump device type is sysdump, refer to the **extendlv** command for more information.

## Callback Commands for System Dumps

A RAS infrastructure component can participate in a system dump. The component uses the RASCD_SET_SDMP_ON command that are passed to the **ras_control** kernel service, and then handles the appropriate commands in the callback routine. Upon the receipt of the callback commands (listed in the first column of the table), the callback routine issues the commands that has _SET in its name (listed in the second column of the table) to perform the actions.

| Callback commands | Commands that are used by callbacks to perform actions | Description |
|---|---|---|
| RASCD_SDMP_ON | RASCD_SET_SDMP_ON | Enables a system dump for the component. |
| RASCD_SDMP_OFF | RASCD_SET_SDMP_OFF | Disables a system dump for the component. |
| RASCD_SDMP_LVL | RASCD_SET_SDMP_LVL | Sets the system dump detail level for this component.<br>The *arg* parameter of the **ras_control** service must be set with a value ranging from 0 through 9. If the level is not set, the component is dumped at the default detail level (CD_LVL_NORMAL). |
| | RASCD_SET_SDMP_CALLONRESTART | Causes the dump to call the callback again if the dump is restarted.<br>If this attribute is not set, the callback is called only once during a system dump to collect data, the location of the returned dump table is saved, and the table is used as if the dump must be restarted. This attribute is assumed if the callback returns an unlimited dump table or requests staging buffer space. |
| | RASCD_SET_SDMP_STAGING | Reserves private staging buffer space.<br>The *arg* parameter must contain the number of bytes to reserve. The maximum value is 32 MB. The **ENOMEM_RAS_SDMP_STAGING** error code is returned if the requested storage is unavailable.<br>You can specify a size of 0 to remove an existing staging buffer. If you need to switch from using a staging buffer to using the shared staging buffer, you must first remove the existing buffer. To remove the existing buffer, issue a RASCD_SET_SDMP_STAGING request with a size of 0, and then issue a RASCD_SET_SDMP_SHARED_STAGING request. |

| Callback commands | Commands that are used by callbacks to perform actions | Description |
| --- | --- | --- |
| | RASCD_SET_SDMP_SHARED_STAGING | Reserves shared staging buffer space. The system has one shared buffer for the system dump. This buffer is as large as the largest shared buffer request.<br>Do not use this buffer to communicate between dump components. The buffer must not contain data that must appear in the dump because the firmware-assisted dump records the data to be copied to the dump device at the next restart. Therefore, the memory that is dumped from the specified address is the last data that existed at the address.<br>Use the buffer only for dump metadata such as dump tables or for keeping the state between the RASCD_SDMP_START and RASCD_SDMP_AGAIN invocations.<br>The *arg* parameter must contain the number of bytes to reserve. The maximum value is 32 MB. The **ENOMEM_RAS_SDMP_STAGING** error code is returned if the requested storage is unavailable.<br>You can specify a size of 0 to remove an existing shared staging-buffer requirement. If you need to switch from using the shared staging buffer to using your own staging buffer, you must first remove the existing requirement. To remove the existing requirement, issue a RASCD_SET_SDMP_SHARED_STAGING request with a size of 0, and then issue a RASCD_SET_SDMP_STAGING request. |
| RASCD_SDMP_START | N/A | Provides data for the system dump.<br>This is equivalent to the DMPRTN_START call that is passed to functions that are registered with the **dmp_add** or **dmp_ctl** kernel service. The *arg* parameter is of the **sdmp_start_t** type. The callback must set its dump table address to the *sdmpst_table* pointer. If the callback returns a negative value, the component is not included in the dump. |
| RASCD_SDMP_AGAIN | N/A | Provides more data for the unlimited dump table.<br>This is equivalent to the DMPRTN_AGAIN call that is passed to functions that are registered with the **dmp_add** or **dmp_ctl** kernel service. The *arg* parameter is of the **sdmp_start_t** type. If the callback returns a negative value, no more data for that component is dumped. The component will not get an RASCD_SDMP_FINISHED call. |
| RASCD_SDMP_FINISHED | N/A | Indicates that the system dump is completed.<br>This is equivalent to the DMPRTN_DONE call that is passed to functions that are registered with the **dmp_add** or **dmp_ctl** kernel service. The RASCD_SDMP_FINISHED call is not issued if a prior RASCD_SDMP_START or RASCD_SDMP_AGAIN call returned a negative return value. |

| Callback commands | Commands that are used by callbacks to perform actions | Description |
|---|---|---|
| RASCD_SDMP_ESTIMATE | N/A | Provides an estimate of how much data will be dumped.<br>The *arg* parameter is of the **sysdump_estimate_t** type. The value that is returned in the *se_value* parameter, is the same as that for the DMPRTN_ESTIMATE invocation in AIX® 5.3. The **sysdump_estimate_t** structure contains the detail level at which to estimate. |
| | RASCD_SET_SDMP_SERIALIO | Enables serialized I/O during dump time. The need for the flag is device-specific. Only the developer of the device can determine whether this flag needs to be set. Use the flag only for the devices that can be on the dump I/O path. Serializing I/O during dump time might degrade dump performance. By default, without the flag, I/O can occur in parallel with function calls of component-dump tables. The action must be performed before the **ras_customize** routine is called, or an error is returned.<br>**Note:** Beginning with AIX® 6.1 with the 6100-02 Technology Level, the **ras_control** kernel service supports the RASCD_SET_SDMP_SERIALIO action flag. |

## Live Dump Facility

Live dumps are small dumps that do not require a system restart. Live dumps replace system dumps when your system is running.

Only the components that are registered for live dumps are dumped. Use the **dumpctrl** command to obtain information about which components are registered for live dumps.

**Important:** The term component in this chapter refers to a component that is specified using the RAS infrastructure (created with the **ras_register** kernel service). Only infrastructure components can be included in a live dump. See "1" on page 370. The **dmp_ctl** and **dmp_add** kernel services only apply to system dumps.

Live dumps can be initiated by software programs or by users with root user authority. Software programs use live dumps as part of recovery actions, or when the runtime error-checking value for the error disposition is ERROR_LIVE_DUMP. See "Initiating Live Dumps from Software Programs" on page 370 and "Sample Kernel Extension" on page 373. If you have root user authority, you can initiate live dumps when a subsystem does not respond or behaves erroneously. For more information about how to initiate and manage live dumps, see the **livedumpstart** and **dumpctrl** commands.

Unlike system dumps, which are written to a dedicated dump device, live dumps are written to the file system. When you install the operating system, a file system is created to contain live dumps. By default, live dumps are placed in the **/var/adm/ras/livedump** directory. You can change the directory using the **dumpctrl** command.

In AIX® Version 6.1 and later, only serialized live dumps are available. A serialized live dump causes a system to be frozen or suspended, when data is being dumped. When the system is frozen, the data is copied into the pinned kernel memory. The data is written to the file system only after the system is unfrozen. A component participating in a live dump must have a callback routine to handle the commands that are passed to the **ras_control** kernel service. See "Callback Commands for Live Dumps" on page 371 for details.

The default live-dump heap size is 64 MB or 1/64 of the size of physical memory, whichever is less. You can change the heap size using the **dumpctrl** command.

Duplicate live dumps that reoccur rapidly are eliminated to prevent system overload and to save file system space. Eliminating duplicate dumps requires periodic (once every 5 minutes) scans of the live dump repository through a cron job.

Each live dump has a data priority. A live dump of **info** priority is for informational purposes, and a live dump of **critical** priority is used to debug a problem. The size of a serialized live dump can be limited by the dump detail level. See “Live Dump Detail Levels” on page 373.

Data structures that are only related to live dumps are listed in the **/usr/include/sys/livedump.h** file.

You can disable all live dumps using the **dumpctrl ldmpoff** command.

## Initiating Live Dumps from Software Programs

A live dump can be initiated from software programs by the kernel or by a kernel extension. Components to be included in the dump must have been registered with the kernel, using the **ras_register** kernel service.

The components must have indicated that they handle live dumps using the RASCD_SET_LDMP_ON **ras_control** service. See “1” on page 370.

To perform a live dump from software programs, follow these steps:

1. Initialize an **ldmp_parms_t** item using the **ldmp_setupparms** kernel service. This step sets up the data structure, filling in all default values including the eye catcher and version fields.
2. Specify infrastructure components using the **dmp_compspec** kernel service, and specify pseudo components using the pseudo components functions.
3. Take the live dump using the **livedump** kernel service.

### *Pseudo components*

A dump pseudo component is a service routine that is used to dump data that is not associated with a component. Such pseudo components are strictly used within a dump.

The following pseudo components are provided.

| Item | Description |
|---|---|
| dmp_eaddr | Dumps memory by effective addresses. |
| dmp_context | Dumps the kernel context. |
| dmp_tid | Dumps a thread. |
| dmp_pid | Dumps a process. |
| dmp_errbuf | Dumps the error logging buffer of the kernel. |
| dmp_mtrc | Dumps entries from the lightweight memory trace buffers. |
| dmp_systrace | Dumps entries from the system trace buffers. |
| dmp_ct | Dumps component trace entries. |

### *Examples*

The following examples shows how to register and include a component for live dumps.

1. In the following example, a component is registered for live dumps with the **ras_register** and **ras_control** kernel services:

```
/* Register a component as dump aware */
rv = ras_register(&Rascb,(char*)Compname,(ras_block_t)0,RAS_TYPE_OTHER,
     "sample component",RASF_DUMP_AWARE,sample_callback,NULL);
if (rv) return(KERROR2ERRNO(rv));

/* Make the component live dump aware. */
```

```
rv = ras_control(Rascb, RASCD_SET_LDMP_ON, 0, 0);
if (rv) return(KERROR2ERRNO(rv));
```

2. In the following example, a component is included in a live dump from software programs:

```
{
    ...
    ldmp_parms_t parms;
    extern ras_block_t Rascb;    /* The ras_block_t from above */

    /* Setup the live dump parms structure. */
    if (ldmp_setupparms(&parm)) {
        /* serious error */
        ...
    }

    /* Each live dump must have a symptom. */
    parms.ldp_symptom = "sample dump";

    /* Include sample_comp in this dump as the failing component. */
    if (dmp_compspec(DCF_FAILING|DCF_BYCB, Rascb, &parm, NULL, NULL)) {
        /* error */
        ...
    }

    /* Add other components and/or pseudo components. */
    ...

    /* Take the dump. */
    if (livedump(&parm)) {
        /* error */
        ...
    }

    ...
}
```

## Callback Commands for Live Dumps

A component participating in a live dump must have a callback routine to handle the following commands that are passed to the **ras_control** kernel service. Upon the receipt of the callback commands (listed in the first column in the table), the callback routine issues the commands that has _SET in its name (listed in the second column in the table) to perform the actions.

| Callback commands | Commands that are used by callbacks to perform actions | Description |
|---|---|---|
| RASCD_LDMP_ON | RASCD_SET_LDMP_ON | Enables a live dump for the component. |
| RASCD_LDMP_OFF | RASCD_SET_LDMP_OFF | Disables a live dump for the component. |
| RASCD_LDMP_LVL | RASCD_SET_LDMP_LVL | Sets the live dump level of the component.<br>The *arg* parameter of the **ras_control** service must be set with a value ranging from 0 through 9. If the level is not set, the component is dumped at the default detail level (CD_LVL_NORMAL). |
| RASCD_LDMP_PREPARE | N/A | Prepares to take a live dump.<br>The callback receives this call when it has been asked to participate in a live dump. The callback uses the **dmp_compspec** kernel service to specify other components to include in the dump, if necessary. The callback can also specify pseudo components. |
| RASCD_LDMP_START | N/A | Dumps data.<br>The callback stores its dump table address in the **ldmpst_table** field of the **ldmp_start_t** data item that is received as an argument.<br>When performing a serialized dump, the callback can use only the services listed in "Kernel Services for a Serialized Dump" on page 372. |

| Callback commands | Commands that are used by callbacks to perform actions | Description |
|---|---|---|
| RASCD_LDMP_AGAIN | N/A | Provides more data for the unlimited dump table (the **cdt_nn_u** type).<br>The return code is similar to that of the RASCD_LDMP_START command, except that if the return value is less than 0, no further data is dumped for the component. Data that was dumped by previous RASCD_LDMP_START and RASCD_LDMP_AGAIN calls appears in the dump.<br>When performing a serialized dump, the callback can use only the services listed in "Kernel Services for a Serialized Dump" on page 372. |
| RASCD_LDMP_FINISHED | N/A | Indicates that the live dump is completed.<br>When performing a serialized dump, the callback can use only the services listed in "Kernel Services for a Serialized Dump" on page 372. |
| RASCD_DMP_PASS_THROUGH | N/A | Passes arbitrary text data to the callback.<br>This command applies to the entire dump domain (there is only one pass-through for the domain containing live and system dumps). You can pass data to a component RASCD_DMP_PASS_THROUGH handler using the **dumpctrl** command. For example, use the `dumpctrl -l foo "pass through text"` command to the RASCD_DMP_PASS_THROUGH handler for the component with the alias of `foo`. |
| RASCD_LDMP_ESTIMATE | N/A | Provides an estimate of how much data will be dumped. The RASCD_LDMP_ESTIMATE call is similar to the RASCD_LDMP_PREPARE call. |

### *Kernel Services for a Serialized Dump*

When data is provided for a serialized live dump, only the following services can be used. The services only apply to the RASCD_LDMP_START, RASCD_LDMP_AGAIN, and RASCD_LDMP_FINISHED calls.

- **ldmp_bufest**, **ldmp_timeleft**, **ldmp_xmalloc**, **ldmp_xmfree**, and **ldmp_errstr**
- **vm_att**, **vm_det**, and **vm_vmid**
- **lqra** and **lra**
- **raschk_safe_read**
- **disable_lock**, **unlock_enable**, **simple_lock**, **simple_lock_try**, and **simple_unlock**
- **i_disable** and **i_enable**
- Lightweight memory trace
- Component Trace
- **sprintf** and **sscanf**
- **printf** (debug only)
- The pinned string functions: **atoi**, **bcmp**, **memccpy**, **memchr**, **memcmp**, **memset**, ***bzero***, **bcopy**, **memcpy**, **memmove**, **strcat**, **strchr**, **strcmp**, **strcpy**, **strcspn**, **strlen**, **strncat**, **strncmp**, **strncpy**, **strpbrk**, **strrchr**, **strspn**, **strstr**, **strtok_r**, and **strtok**

A component can specify any data to be dumped, however, in a serialized dump, only memory-resident data is dumped. For a system dump, each data area in the dump has an associated bit map that indicates whether the data is in the dump, or the data cannot be included in the dump because it is not memory resident. While participating in a serialized live dump, a component must not directly refer to any storage that is not memory resident. To ensure safe access to data, use the **raschk_safe_read** kernel service with the **RAS_SR_NOPAGEIN** flag.

Do not use the system trace. If a system trace buffer fills, entries are lost until the system is unfrozen. Use the lightweight memory trace and component trace.

## Live Dump Detail Levels

In a serialized live dump, the amount of information that is dumped for a component is limited according to the dump detail level. Unless otherwise specified, the dump detail level of a component is CD_LVL_NORMAL.

The detail level ranges from CD_LEVEL_0 through CD_LEVEL_9. Three levels are used frequently: CD_LVL_MINIMAL (CD_LEVEL_1), CD_LVL_NORMAL (CD_LEVEL_3), and CD_LVL_DETAIL (CD_LEVEL_7). A component can query the value using the **rasrb_ldmp_level**(**rasb**) service, and can set the detail level with the **RASCD_SET_LDMP_LVL** command in the **ras_control** kernel service.

The following table shows the data limits for a component. If the component exceeds the limit, its data is truncated and only the data entries before the one that causes the limit to be exceeded are dumped.

| Live dump detail level | Maximum size |
|---|---|
| < CD_LVL_NORMAL | 2 MB |
| ≥ CD_LVL_NORMAL & < CD_LVL_DETAIL | 4 MB |
| ≥ CD_LVL_DETAIL & < CD_LEVEL_9 | 8 MB |
| CD_LEVEL_9 | unlimited |

## Sample Kernel Extension

The following sample is a kernel extension that takes a live dump and a system dump.

The **sample_callback** function takes a live dump and a system dump using the commands that are sent by the system and passed to the **ras_control** kernel service. The sample only shows the handling of the dump commands. Normally, the callback must handle component trace and error checking commands.

```
/*
 * This sample creates a component, makes it dump-aware, and handles both live
 * and system dump.
 */
#include <sys/types.h>
#include <sys/syspest.h>
#include <sys/uio.h>
#include <sys/processor.h>
#include <sys/systemcfg.h>
#include <sys/malloc.h>
#include <sys/ras.h>
#include <sys/livedump.h>
#include <sys/kerrnodefs.h>
#include <sys/eyec.h>
#include <sys/raschk.h>
#include <sys/param.h>

/* Component name and handle */
const char Compname[] = "sample_comp";
ras_block_t Rascb=NULL;

/*
 * The sample data dumped consists of a header plus an unlimited number
 * of chained control blocks.
 * The header is dumped first.
 * Because we are using an unlimited dump table, the callback then gets an
 * "AGAIN" call to dump the rest of the data, the control blocks.
 * We'll dump these 3 at a time until finished.
 * The staging area is used for the dump table.
 */
#define NENTRIES 3
/* Staging area size for dumping the header. */
#define SZ1 (sizeof(struct cdt_nn_head) + sizeof(struct cdt_entry_u))
/* Staging area size for 3 control blocks. */
#define SZ2 (sizeof(struct cdt_nn_head) + DMP_DUL_SIZE(NENTRIES))
/* Staging buffer size is the MAX of SZ1 and SZ2 above. */
size_t Sbufsz;
/*
 * To estimate the dump table space, we need SZ1 plus the unlimited entry
```

```
 * for how many control blocks we have at dump time.
 */
#define TBLESTSZ(n) (SZ1 + DMP_DUL_SIZE(n))

/*
 * This is the sample data.
 * It would normally be protected with a lock, however, that is not shown here.
 */
typedef struct sample_cb {
    eye_catch8b_t    scb_eyec;    /* must be EYEC_SCB */
    struct sample_cb *scb_next;    /* list ptr, terminated with
                        INVALID_SCB_PTR */
    long        scb_flags;
    long        scb_fld1;
} sample_cb_t;
#define EYEC_SCB    __EYEC8('s','a','m','p','l','A','B','C')
#define INVALID_SCB_PTR    INVALID_PTR(EYEC_SCB)

typedef struct sample_hdr {
    eye_catch8b_t    sh_eyec;    /* must be EYEC_SH */
    long        sh_flags;
    sample_cb_t    *sh_cbp;    /* ptr to first cb */
    long        sh_fld1;
} sample_hdr_t;
#define EYEC_SH        __EYEC8('s','a','h','d','r','A','B','C')
sample_hdr_t Sample_hdr = {EYEC_SH, 0, INVALID_SCB_PTR, 0};

kerrno_t sample_callback(ras_block_t cb, ras_cmd_t cmd, void *arg, void *priv);
int alloc_sample(int n);
void free_sample();
static sample_cb_t *get_scbp(sample_cb_t *addr, dumpid_t id, int ldmpflag);

/*
 * Entry point called when this kernel extension is loaded.
 *
 * Input:
 *     cmd - unused (typically 1=config, 2=unconfig)
 *     uiop - points to the uio structure.
 */
int
sampleext(int cmd, struct uio *uiop)
{
    kerrno_t rv = 0;
    int rc;

    /* cmd should be 1 or 2 */
    if (cmd == 2) {
        /* Unloading */
        if (Rascb) ras_unregister(Rascb);
        free_sample();
        return(0);
    }
    if (cmd != 1) return(EINVAL);

    /* Set up local variables. */
    Sbufsz = MAX(SZ1, SZ2);

    /* The extension is being loaded, set up the sample data. */
    rc = alloc_sample(NENTRIES);
    if (rc) return(rc);

    /* Register the component as dump aware */
    rv = ras_register(&Rascb,
        (char*)Compname,
        (ras_block_t)0,
        RAS_TYPE_OTHER,
        "sample component",
        RASF_DUMP_AWARE,
        sample_callback,
        NULL);
    if (rv) return(KERROR2ERRNO(rv));

    /* Make the component system and live dump aware. */
    rv = ras_control(Rascb, RASCD_SET_SDMP_ON, 0, 0);
    if (rv) return(KERROR2ERRNO(rv));
    rv = ras_control(Rascb, RASCD_SET_LDMP_ON, 0, 0);
    if (rv) return(KERROR2ERRNO(rv));

    /*
     * System dump staging buffer space must be set up before a
     * system dump occurs.
     * Staging buffer space for live dumps is set up by the callback at
```

```
     * live dump time.
     */
    rv = ras_control(Rascb, RASCD_SET_SDMP_STAGING,
             (void*)Sbufsz, 0);
    if (rv) return(KERROR2ERRNO(rv));

    /*
     * The component must be customized.
     * It uses the default level, CD_LVL_NORMAL.
     */
    rv = ras_customize(Rascb);
    if (rv) return(KERROR2ERRNO(rv));

    return(0);
}

/*
 * Sample callback that is called for live and system dumps.
 *
 * The data to dump consists of a header and
 * control blocks.  The data is dumped using an unlimited dump table.
 * The header is dumped first, followed by the control blocks, dumped 3 at a
 * time until all have been dumped.
 *
 * Input:
 *   cb   - Contains the component's ras_block_t
 *   cmd  - ras_control command
 *   arg  - command argument
 *   priv - private data, unused
 */
kerrno_t
sample_callback(ras_block_t cb, ras_cmd_t cmd, void *arg, void *priv)
{
    kerrno_t rv = 0;
    sample_cb_t *cbp, **wkptr;

    switch(cmd) {
    /* Live dump */
    case RASCD_LDMP_ON: {
        /* Turn live dump on. */
        rv = ras_control(cb, RASCD_SET_LDMP_ON, 0, 0);
        break;
        }
    case RASCD_LDMP_OFF: {
        /* Turn live dump off. */
        rv = ras_control(cb, RASCD_SET_LDMP_OFF, 0, 0);
        break;
        }
    case RASCD_LDMP_LVL: {
        /* Set the detail level at which this component will dump. */
        rv = ras_control(cb, RASCD_SET_LDMP_LVL, arg, 0);
        break;
        }
    case RASCD_LDMP_ESTIMATE: /* fall through */
    case RASCD_LDMP_PREPARE:{
        /*
         * An estimate, as a prepare, is done in the same way.
         * The estimate is received if the livedumpstart command is used
         * and the -e flag is specified.
         * The prepare is received when the component is participating
         * in a live dump.  The prepare call is used to request
         * staging buffer space and provide an estimate of the amount
         * of data to be dumped. The sample also requests that the component
         * trace be dumped at this time.
         */
        ldmp_prepare_t *p = (ldmp_prepare_t*)arg;
        int n = 0;
        /* Staging buffer used for dump table */
        p->ldpr_sbufsz = Sbufsz;
        /* Data size - need # cbs */
        for (cbp=get_scbp(Sample_hdr.sh_cbp, p->ldpr_dumpid, 1), n=0;
             cbp!=INVALID_SCB_PTR;
             cbp=get_scbp(cbp->scb_next, p->ldpr_dumpid, 1), n++);
        p->ldpr_datasize = TBLESTSZ(n) + sizeof(Sample_hdr) +
                   n*sizeof(sample_cb_t);
        /* Dump all of our component trace. */
        rv = dmp_ct(0, p, "", Rascb, 0);
        /*
         * If an error occurred, ldmp_errstr() puts the message in
         * the dump. The sample returns 0, so the dump
         * proceeds normally.
         * If the sample returned a value > 0,
```

```
         * livedump() would put a generic error containing the
         * return value in the dump.
         */
        if (rv) {
            char str[40];
            sprintf(str, "dmp_ct returned 0x%lx.\n", rv);
            rv = 0;
            (void)ldmp_errstr(p->ldpr_dumpid, Rascb, str);
        }
        break;
        }
    case RASCD_LDMP_START:{
        /*
         * This is received to provide the dump table.
         * Because the table is an unlimited table, subsequent
         * RASCD_LDMP_AGAIN calls will be received.
         */
        ldmp_start_t *p = (ldmp_start_t*)arg;
        struct cdt_nn_head_u *hp;
        struct cdt_entry_u *ep;
        /* The dump table goes in the staging area */
        hp = (struct cdt_nn_head_u*)p->ldmpst_buffer;
        ep = (struct cdt_entry_u*)((struct cdt_nn_u*)hp)->cdtnu_entry;
        /* Set up cdt_nn_head_u */
        hp->cdtnu_magic = DMP_MAGIC_NU;
        hp->cdtnu_nentries = 1;
        /* Set up cdt_entry_u */
        ep->du_magic = DMP_MAGIC_UD;
        strcpy(ep->du_name, "header");
        ep->du_len = sizeof(Sample_hdr);
        ep->du_ptr = &Sample_hdr
        ep->du_segval = DUMP_GEN_SEGVAL;
        ep->du_xmemp = NULL;
        p->ldmpst_table = hp;
        /*
         * There is a work area in the ldmp_prepare_t and ldmp_start_t
         * data areas for use in keeping the state across dump calls,
         * generally between RASCD_LDMP_START and RASCD_LDMP_AGAIN
         * commands.
         * In our case, we'll keep a pointer to the next cb to dump,
         * NULL initially.
         */
        wkptr = (sample_cb_t**)p->ldmpst_wk;
        *wkptr = NULL;
        break;
        }
    case RASCD_LDMP_AGAIN:{
        /*
         * This is similar to the  RASCD_LDMP_START command, but is received to dump
         * subsequent data for an unlimited dump table.
         */
        int i;
        ldmp_start_t *p = (ldmp_start_t*)arg;
        struct cdt_nn_head_u *hp;
        struct cdt_entry_ul *up;
        sample_cb_t *cbp;
        /* The dump table goes in the staging area */
        hp = (struct cdt_nn_head_u*)p->ldmpst_buffer;
        up = (struct cdt_entry_ul*)((struct cdt_nn_u*)hp)->cdtnu_entry;
        /* Point to the first/next cb */
        wkptr = (sample_cb_t**)p->ldmpst_wk;
        /* For the first AGAIN call, cbp will be Sample_hdr.sh_cbp. */
        cbp = (*wkptr)? *wkptr: Sample_hdr.sh_cbp;
        /* Validate the pointer. */
        cbp = get_scbp(cbp, p->ldmpst_dumpid, 1);
        /* Set up cdt_nn_head_u */
        hp->cdtnu_magic = DMP_MAGIC_NU;
        hp->cdtnu_nentries = 1;
        /* Set up cdt_entry_ul */
        up->dul_magic = DMP_MAGIC_UL;
        strcpy(up->dul_name, "cb");
        up->dul_nentries = 0;
        up->dul_len = sizeof(sample_cb_t);
        /* Dump up to 3, NENTRIES, control blocks. */
        for (i=0; iscb_next, p->ldmpst_dumpid, 1)) {
            up->dul_entry[i].dle_vmhandle = DUMP_GEN_SEGVAL;
            up->dul_entry[i].dle_ptr = cbp;
            up->dul_nentries++;
        }
        /* Save address of next cb. */
        *wkptr = cbp;
        /* Set the table address to NULL when finished. */
```

```
        p->ldmpst_table = (!up->dul_nentries)? NULL: p->ldmpst_buffer;
        break;
        }
    case RASCD_LDMP_FINISHED:
        /* Nothing to do here. */
        break;

    /* System dump */
    case RASCD_SDMP_ON: {
        /* Turn system dump on. */
        rv = ras_control(cb, RASCD_SET_SDMP_ON, 0, 0);
        break;
        }
    case RASCD_SDMP_OFF: {
        /* Turn system dump off. */
        rv = ras_control(cb, RASCD_SET_SDMP_OFF, 0, 0);
        break;
        }
    case RASCD_SDMP_LVL: {
        /* Set the detail level at which this component will dump. */
        rv = ras_control(cb, RASCD_SET_SDMP_LVL, arg, 0);
        break;
        }
    case RASCD_SDMP_ESTIMATE:{
        sysdump_estimate_t *p = (sysdump_estimate_t*)arg;
        int n;
        /* Data size - need # cbs */
        for (cbp=get_scbp(Sample_hdr.sh_cbp, 0, 0), n=0;
             cbp!=INVALID_SCB_PTR;
             cbp=get_scbp(cbp->scb_next, 0, 0), n++);
        p->se_value = TBLESTSZ(n) + sizeof(Sample_hdr) +
                      n*sizeof(sample_cb_t);
        break;
        }
    case RASCD_SDMP_START:{
        /*
         * This is received to provide the dump table.
         * Since the table is an unlimited table, subsequent
         * RASCD_SDMP_AGAIN calls will be received.
         */
        sdmp_start_t *p = (sdmp_start_t*)arg;
        struct cdt_nn_head_u *hp;
        struct cdt_entry_u *ep;
        /* The dump table goes in the staging area */
        hp = (struct cdt_nn_head_u*)p->sdmpst_buffer;
        ep = (struct cdt_entry_u*)((struct cdt_nn_u*)hp)->cdtnu_entry;
        /* Set up cdt_nn_head_u */
        hp->cdtnu_magic = DMP_MAGIC_NU;
        hp->cdtnu_nentries = 1;
        /* Set up cdt_entry_u */
        ep->du_magic = DMP_MAGIC_UD;
        strcpy(ep->du_name, "header");
        ep->du_len = sizeof(Sample_hdr);
        ep->du_ptr = &Sample_hdr
        ep->du_segval = DUMP_GEN_SEGVAL;
        ep->du_xmemp = NULL;
        p->sdmpst_table = hp;
        /*
         * There is a work area in the sdmp_start_t data area
         * for use in keeping the state across dump calls, generally
         * between RASCD_SDMP_START and RASCD_SDMP_AGAIN
         * commands.
         * In our case, we'll keep a pointer to the next cb to dump,
         * NULL initially.
         */
        wkptr = (sample_cb_t**)p->sdmpst_wk;
        *wkptr = NULL;
        break;
        }
    case RASCD_SDMP_AGAIN:{
        /*
         * This is similar to RASCD_SDMP_START, but is received to dump
         * subsequent data for an unlimited dump table.
         */
        int i;
        sdmp_start_t *p = (sdmp_start_t*)arg;
        struct cdt_nn_head_u *hp;
        struct cdt_entry_ul *up;
        sample_cb_t *cbp;
        /* The dump table goes in the staging area */
        hp = (struct cdt_nn_head_u*)p->sdmpst_buffer;
        up = (struct cdt_entry_ul*)((struct cdt_nn_u*)hp)->cdtnu_entry;
```

```
        /* Point to the first/next cb */
        wkptr = (sample_cb_t**)p->sdmpst_wk;
        /* For the first AGAIN call, cbp will be Sample_hdr.sh_cbp. */
        cbp = (*wkptr)? *wkptr: Sample_hdr.sh_cbp;
        /* Validate the pointer. */
        cbp = get_scbp(cbp, 0, 0);
        /* Set up cdt_nn_head_u */
        hp->cdtnu_magic = DMP_MAGIC_NU;
        hp->cdtnu_nentries = 1;
        /* Set up cdt_entry_ul */
        up->dul_magic = DMP_MAGIC_UL;
        strcpy(up->dul_name, "cb");
        up->dul_nentries = 0;
        up->dul_len = sizeof(sample_cb_t);
        /* Dump up to 3, NENTRIES, control blocks. */
        for (i=0; iscb_next, 0, 0)) {
            up->dul_entry[i].dle_vmhandle = DUMP_GEN_SEGVAL;
            up->dul_entry[i].dle_ptr = cbp;
            up->dul_nentries++;
        }
        /* Save address of next cb. */
        *wkptr = cbp;
        /* Set the table address to NULL when finished. */
        p->sdmpst_table = (!up->dul_nentries)? NULL: p->sdmpst_buffer;
        break;
        }
    case RASCD_SDMP_FINISHED:
        /* Nothing to do here. */
        break;

    case RASCD_DMP_PASS_THROUGH:{
        /* pass through */
        printf("%s\n", arg);
        break;
        }
    default: {
        printf("bad ras_control command.\n");
        rv = EINVAL_RAS_CONTROL_BADCMD;
        }
    }

    return(rv);
}

/*
 * Allocate sample data
 *
 * Input:
 *   n - number of sample cbs to allocate.
 *
 * Returns:
 *   0 - success
 *   errno - errno from failure.
 */
int
alloc_sample(int n)
{
    sample_cb_t *cbp, *prev_cbp=NULL;

    /* Allocate n cbs */
    while (n--) {
        cbp = xmalloc(sizeof(*cbp), 3, kernel_heap);
        if (!cbp) {
            free_sample();
            return(ENOMEM);
        }
        if (!prev_cbp) Sample_hdr.sh_cbp = cbp;
        else prev_cbp->scb_next = cbp;
        cbp->scb_eyec = EYEC_SCB;
        cbp->scb_next = INVALID_SCB_PTR;
        prev_cbp = cbp;
    }

    return(0);
}

/*
 * Free sample data
 */
void
free_sample()
{
```

```
    sample_cb_t *cbp = Sample_hdr.sh_cbp;

    /* Validate cbp */
    cbp = get_scbp(cbp, 0, 0);

    while(cbp != INVALID_SCB_PTR) {
        sample_cb_t *save_cbp = cbp;
        cbp = get_scbp(cbp->scb_next, 0, 0);
        xmfree(save_cbp, kernel_heap);
    }
}

/*
 * Validate the cb at the supplied address.
 * This ensures we won't get an exception for a bad scb_next ptr.
 *
 * Input:
 *   addr -   - address to read from
 *   id   -   - dump ID
 *   ldmpflag - 1 if this is a live dump.
 *
 * Returns:
 *   The pointer at the address.
 *   INVALID_SCB_PTR if the memory at addr is bad.
 */
static sample_cb_t *
get_scbp(sample_cb_t *addr, dumpid_t id, int ldmpflag)
{
    sample_cb_t scb;
    char str[80];

    /* Just return if addr is the terminating value. */
    if (addr == INVALID_SCB_PTR) return(addr);

    /* Carefully get the storage at addr. */
    if (raschk_safe_read(addr, &scb, sizeof(scb), RAS_SR_NOFAULT)) {
        /* addr is bad. */
        if (ldmpflag) {
            sprintf(str, "The scb address 0x%lx is bad.\n", addr);
            (void)ldmp_errstr(id, Rascb, str);
        }
        addr = INVALID_SCB_PTR;
    }
    else {
        /* Validate the control block */
        if (scb.scb_eyec != EYEC_SCB) {
            /* cb appears bad. */
            if (ldmpflag) {
                sprintf(str, "cb at 0x%lx is invalid, eyec = 0x%lx.\n",
                    addr, scb.scb_eyec);
                (void)ldmp_errstr(id, Rascb, str);
            }
            addr = INVALID_SCB_PTR;
        }
    }

    return(addr);
}
```

## Component Trace Facility

Component Trace (CT) is an important First Failure Data Capture (FFDC) and Second Failure Data Capture (SFDC) tool available to the kernel, kernel extensions, and device drivers. With the Component Trace facility, a component can capture trace events to aid both debugging and system analysis and can provide focused trace data on larger server systems.

Component Trace uses mechanisms similar to system trace. Existing **TRCHK** and **TRCGEN** macros can be replaced with **CT** macros to trace into system trace buffers and private buffers of memory-trace mode.

If recorded, Component Trace events can be retrieved with the **ctctrl** command. Extraction with the **ctctrl** command is relevant only to in-memory tracing. Component Trace events can also be present in a system trace. You can use the **trcrpt** command for Component Trace and system trace to process the events.

Component trace entries can be traced to the private buffer of the component, the lightweight memory trace, the system trace, or any combination of these destinations. The destination is governed by flags specified in the **CT_HOOK**x and **CT_GEN** macros. The **MT_COMMON** flag causes the entry to be traced

into the common, lightweight-memory-trace buffer. The **MT_RARE** flag causes the entry to go to the rare, lightweight-memory-trace buffer. Do not specify both the **MT_COMMON** and **MT_RARE** flags. The **MT_PRIV** flag traces the entry into the private buffer of the component. The **MT_SYSTEM** flag puts the entry into a system trace if system trace is active.

Generic trace entries, which are traced with the **CT_GEN** macro, cannot be traced into the lightweight memory trace.

In the memory trace mode, you have the choice for each component, at initialization, to store their trace entries either in a component private buffer or in one of the memory buffers managed by the lightweight memory trace. When entries are stored in a lightweight-memory-trace buffer, the memory type (common or rare) is chosen for each trace entry.

The private buffer of the component is a pinned memory buffer that can be allocated by the framework at the component registration or later. The buffer is only attached to this component. You can dynamically change the buffer size with the **CT** API. Administrators can dynamically change the buffer size using the **ctctrl** command.

Private buffers and lightweight memory buffers are used in a circular mode, that is, if the buffer is full, the last trace entries overwrite the first one.

For each component, the serialization of the buffers can be managed either by the component (by the component owner) or by the Component Trace framework. This serialization policy is chosen at registration and cannot be changed during the life of the component.

The system trace mode is another function that is provided by Component Trace. When a component is traced using a system trace, each trace entry is sent to the current system trace. In this mode, Component Trace acts as a front-end filter for the existing system trace. By setting the level of the system trace, a component can control which trace hooks enter the system trace buffer.

## Component Trace Modes

Component Trace has two modes that can be used simultaneously - system trace mode and memory trace mode.

**system trace mode**

The system trace mode sends trace entries to the existing system trace. The following settings can be changed:

| Setting | Default setting |
|---|---|
| on or off | The mode is on. |
| trace level | The default level of system trace is CT_LVL_NORMAL (level 3). |

**memory trace mode**

The memory trace mode stores the trace entries in a memory buffer. The buffer is either private to the component or to a per-processor memory buffer that is dedicated to the lightweight memory trace. The following settings can be changed:

| Setting | Default setting |
|---|---|
| on or off | The mode is off. |
| serialization policy | The recording of trace entries is serialized by the framework. |
| tracing status | Tracing is suspended. |
| the size of the private buffer | 0 |
| trace level | The default level of memory trace is CT_LVL_NORMAL (level 3). |

## Using the Component Trace Facility

This overview contains steps to use the component trace facility from initializing a component trace to managing the trace levels.

### *Initializing a Component Trace*

Component Trace is initialized in the process environment, preferably during the initialization of the driver or the kernel subsystem for which the component will be used.

The initialization must be done for all components and subcomponents. Initialization is done in three steps.

*Registration*
A component is registered into the Component Trace framework by calling the **ras_register** kernel service.

The following code is an example of registering a parent (base) component by using the **ras_register** kernel service.

```
ras_block_t rasb_eth;

kerrno_t err;
    ...
err = ras_register(&rasb_eth, "ethernet", NULL, RAS_TYPE_NETWORK_ETHERNET, "All ethernet
devices",
 RASF_TRACE_AWARE, eth_callback, NULL);
```

The example creates and registers a base component named `ethernet` of the `network` type and the `ethernet` subtype (the fourth argument). The *type/subtype* field is used to categorize a component. You can get a list of available types and subtypes in the **sys/ras_base.h** file. The `rasb_eth` argument, which is the component identifier (or reference), is used for all actions to apply to the component. The third argument is set to NULL, indicating that this component is a base component with no parent. See "Callback Routine" on page 383 for details about the callback routine (the seventh argument). In the example, no *callback_data* area (the last argument) is specified. If one is specified, it is passed through the callback routine. For more information, see the **ras_register** kernel service.

*Changing Component Trace Properties*
Before customizing, you might want to change some of the default Component Trace properties.

You can change the following settings:

- To activate the memory trace mode, use the following code:

  ```
  /* set a buffer size (default size is 0) */

  err = ras_control(rasb_eth, RASCT_SET_MEMBUFSIZE, size, 0);

  /* allocate the private buffer */
  err = ras_control(rasb_eth, RASCT_SET_ALLOC_BUFFER, 0, 0);

  /* activate memory trace mode */
  err = ras_control(rasb_eth, RASCT_SET_MEMTRC_RESUME, 0, 0);
  ```

  If you activate memory trace mode without allocating a private buffer, leave out the RASCT_SET_MEMBUFSIZE and RASCT_SET_ALLOC_BUFFER calls. This is useful if you want to just send events to the system trace or to the lightweight memory trace. Failure in RAS routines should not cause a driver or kernel subsystem to abort initialization.
- To use component serialization rather than infrastructure serialization, use the following code:

  ```
  err = ras_control(rasb_eth, RASCT_SET_CT_SERIALIZED, 0, 0);
  ```

  You cannot change this setting after customization.
- To change the level of trace to the minimal value for system trace mode, use the following code:

```
err = ras_control(rasb_eth, RASCT_SET_SYSTRC_LVL, CT_LVL_MINIMAL, 0);
```

You can get the definitions of available levels in the **ras_trace.h** file.

- To change the level of trace to a detailed value for memory trace mode, use the following code:

```
err = ras_control(rasb_eth, RASCT_SET_MEMTRC_LVL, CT_LVL_DETAIL, 0);
```

For more information, see the **ras_control** kernel service.

*Customization*
The customization step enables component tracing for the specified component and retrieves any saved component settings. Customization is useful for keeping settings of components after a system restart.

The customization step is mandatory to put the component into a usable state. Before this call, no tracing is active even if either trace mode has been activated. You can customize the `ethernet` component as follows, continuing the examples in the "Changing Component Trace Properties" on page 381 section:

```
err = ras_customize(rasb_eth);
```

After this call, the initialization of your component with the Component Trace framework is complete, and tracing is active unless a persistent value has been set to indicate that the Component Trace is off.

For example, if the **ctctrl** command was used to set Component Tracing to be persistently off, any call to the **ras_customize** kernel service results in the component trace of the calling component being turned off. In this case, unless the **ctctrl** command is used to set Component Trace to be persistently on, all requests to turn on or resume component tracing are denied.

### *Unregistering a Component*
If the registered component is a driver, a kernel extension or a subsystem that can be stopped. Unregister your component at stop or unload time.

A component cannot be unregistered from the framework if it has subcomponents. Therefore, subcomponents must be unregistered first. Moreover, the memory trace mode must be stopped.

```
ras_control(rasb_eth, RASCT_SET_MEMTRC_SUSPEND, 0, 0);  /* might need to be serialized */
ras_unregister(rasb_eth);
```

The **ras_unregister** call must be done from the process environment. No other driver or subsystem operations should occur during **ras_unregister** calls. For more information, see the **ras_unregister** kernel service.

### *Tracing Events into the Component Trace Private Buffer*
This step is to trace events into the Component Trace private buffer.

For more information about how to trace events into the Component Trace private buffer, see the **CT_HOOKx** and **CT_GEN** macros.

### *Controlling Component Trace for Your Component*
This step is to control the Component Trace for your component.

For more information about how to control Component Trace and dump Component Trace buffers, see the **ctctrl** command.

### *Managing Trace Levels*
The trace levels are set to a value of -1 if the mode is suspended or off.

To access the memory trace mode level of a component, use the following code:

```
rasrb_trace_memlevel(rasb_eth)
```

To access the system trace mode level, use the following code:

```
rasrb_trace_syslevel(rasb_eth)
```

You can use the `CT_TRCON(rasb_eth, level)` macro to know if the trace entries traced at that level are recorded in one or both of the trace modes (system or memory).

The system mode is considered to be on if system trace is running and the system trace mode is on for the specified component.

## Callback Routine

For components that are registered for Component Trace, a callback routine is mandatory. The routine is called by the Component Trace framework to inform registered components about an event that might require action.

The component can accept or reject a request by either performing or not performing the appropriate commands that the **ras_control** kernel service passes to the callback.

The callback routine is called when a new setting is requested using the **ctctrl** user command (except for DR events). The following commands must be handled to ensure the correct operation of the **ctctrl** command:

| Command | Description |
|---|---|
| **RASCT_MEMTRC_ON** | Sets the memory trace mode on for this component. This command can only be called from the process environment. The **RASCT_MEMTRC_ON** command is passed to the component through the callback routine. The component must perform the **RASCT_SET_ALLOC_BUFFER** command to allocate a buffer and perform the **RASCT_SET_MEMTRC_RESUME** command to enable memory trace mode. |
| **RASCT_MEMTRC_OFF** | Sets the memory trace mode off for this component. This command can only be called from the process environment. The **RASCT_MEMTRC_OFF** command is passed to the component through the callback routine. The component must perform the **RASCT_SET_MEMTRC_SUSPEND** command to disable memory trace mode followed by the **RASCT_SET_FREE_BUFFER** command to free the buffer. If the component is not framework-serialized, the component must serialize the **RASCT_SET_MEMTRC_SUSPEND** call.<br><br>When the **RASCT_MEMTRC_OFF** command is received, the component must free its private buffer. Otherwise, the framework will free the private buffer. The **RASCT_MEMTRC_SUSPEND** command is used instead of the **RASCT_MEMTRC_OFF** command if the buffer is needed again. |
| **RASCT_MEMTRC_SUSPEND**<br>**RASCT_SET_MEMTRC_SUSPEND** | Suspends the memory trace mode for this component by turning this mode off without freeing any private buffer. This command can only be called from the process environment. The **RASCT_MEMTRC_SUSPEND** command is passed to the component through the callback routine. The component must perform the **RASCT_SET_MEMTRC_SUSPEND** command to apply the settings. If the component is not framework-serialized, the component must serialize the **RASCT_SET_MEMTRC_SUSPEND** call. |
| **RASCT_MEMTRC_RESUME** | Starts the memory trace mode for this component to record trace events. |
| **RASCT_SET_MEMTRC_RESUME** | The command is usually used after tracing has suspended with a **RASCT_SET_MEMTRC_SUSPEND** command. This command can only be called from the process environment. The **RASCT_MEMTRC_RESUME** command is passed to the component through the callback routine. The component must perform a **RASCT_SET_MEMTRC_RESUME** control command to apply the settings. This command fails if a resize or save operation is in progress, or if a buffer is required but not allocated. |

| Command | Description |
| --- | --- |
| **RASCT_SYSTRC_ON**<br>**RASCT_SET_SYSTRC_ON** | Sets the system trace mode on for this component. This command can be called in both interrupt and process environments. The **RASCT_SYSTRC_ON** command is passed to the component through the callback routine. The component musts perform the **RASCT_SET_SYSTRC_ON** control command to apply the settings. |
| **RASCT_SYSTRC_OFF**<br>**RASCT_SET_SYSTRC_OFF** | Sets the system trace mode off for this component. This command can be called in both interrupt and process environments. The **RASCT_SYSTRC_OFF** command is passed to the component through the callback routine. The component must perform the **RASCT_SET_SYSTRC_OFF** control command to apply the settings. |
| **RASCT_MEMTRC_LVL**<br>**RASCT_SET_MEMTRC_LVL** | Changes the level of trace for the memory trace mode. The level of trace must be passed through the *arg* parameter of the **ras_control** kernel service. This command can be called in both interrupt and process environments. The **RASCT_MEMTRC_LVL** command is passed to the component through the callback routine. The component must perform the **RASCT_SET_MEMTRC_LVL** control command to apply the settings. |
| **RASCT_SYSTRC_LVL**<br>**RASCT_SET_SYSTRC_LVL** | Changes the level of trace for the system trace mode. The level of trace must be passed through the *arg* parameter. This command can be called in both interrupt and process environments. The **RASCT_SYSTRC_LVL** command is passed to the component through the callback routine. The component must perform the **RASCT_SET_SYSTRC_LVL** control command to apply the settings. |
| **RASCT_MEMBUFSIZE** | Changes the size of the private buffer. The new size must be given in bytes through the *arg* parameter. This command can be called only in the process environment. The **RASCT_MEMBUFSIZE** command is passed to the component; the component must call the **RASCT_SET_MEMBUFSIZE** control command to apply the property. The **RASCT_SET_MEMBUFSIZE** command can be called regardless of whether a private buffer is allocated. But first, the memory trace mode must be suspended with the **RASCT_SET_MEMTRC_SUSPEND** command if the component is already on. Also, if a private buffer was allocated, the last traced entries are preserved. The exact amount depends on the new buffer size. Finally, the **RASCT_SET_ALLOC_BUFFER** command must be used. The command is necessary if no private buffer was allocated previously, but it can be used in either case. If a nonzero value is requested, a minimum value of 2 times the size of a full trace entry (with five data words) is required.<br><br>The **RASCT_MEMBUFSIZE** command is called if the `ctctrl memtracebufsize=size` user command is performed. The component might want to keep track of the suspend or resume state in order not to resume the memory trace mode if this mode was not on before the call. |
| **RASCT_SET_ALLOC_BUFFER** | Allocates the private memory buffer. This command can be called only from the process environment. This command fails if the memory trace mode is not suspended. |
| **RASCT_SET_FREE_BUFFER** | Frees the private memory buffer. This command can be called only from the process environment. This command fails if the memory trace mode is not suspended, or if a resize or save operation is in progress. |
| **RASCT_SET_CT_SERIALIZE** | Sets the serialization policy for the component. If the value of the *arg* parameter is TRUE (nonzero), the framework serializes buffer access. Otherwise, it must be serialized by the component. The default value is TRUE. This property must be set at initialization phase, and it can not be changed after the customization phase. Moreover, only the component can set the property. This command can be called in both interrupt and process environments. |
| **RASCT_DR_MEM** | Informs the component of add-memory or remove-memory operation. The command is passed from the process environment. The total amount of system memory that is added or removed is passed to the component through the *arg* field of the **ras_callback** function. A component can resize its private buffer at this time. |

| Command | Description |
| --- | --- |
| **RASCT_PASS_THROUGH** | Passes string data to the callback function. The **ctctrl** command can be used to pass a string (for example, `ctctrl -c socket "passthrough string"`). The string is passed with the *arg* parameter. The component can use this command to perform some specific actions. It returns the **EINVAL** error code for all unavailable commands. This command can be used only from the process environment. |
| **RASCT_GETBUFFER** | Retrieves a copy of the current contents of the private buffer. The memory trace mode must be disabled during the copy to preserve the integrity of the copy. If the memory trace mode is enabled, this command attempts to issue the **RASCT_MEMTRC_SUSPEND** command, retrieve the buffer contents, and then issue the **RASCT_MEMTRC_RESUME** command. The *arg* and *size* parameters must contain the pointer and the size of the provided buffer for the copy. This command can be called only from the process environment. |

By convention, the callback must return a value of 0, if it accepts the setting and all **ras_control** commands run by the callback return successfully. Otherwise the callback must return an error value.

All **RASCT_SET_ALLOC_BUFFER**, **RASCT_SET_FREE_BUFFER**, and **RASCT_SET_MEMBUFSIZE** commands must be called when memory trace mode is disabled (suspended) to ensure that the buffer is not used for tracing (by **CT_HOOK** calls). The **RASCT_SET_MEMTRC_SUSPEND** command must then be serialized the same way as the **CT_HOOKx** and **CT_GEN** calls.

Components can update some internal values or keep track of the context modification using the **callback_data** area. The **callback_data** area that is passed to the callback routine is the one given at the registration time.

## Error Logging

The error facility records device-driver entries in the system error log. These error log entries record any software or hardware failures that need to be available either for informational purposes or for fault detection and corrective action.

The device driver, using the **errsave** kernel service, adds error records to the **/dev/error** special file.

The **errdemon** daemon picks up the error record and creates an error log entry. When you access the error log either through SMIT (System Management Interface Tool) or with the **errpt** command, the error record is formatted according to the error template in the error template repository and presented in either a summary or detailed report.

Before initiating the error logging process, determine what services are available to developers, and what services are available to the customer, service personnel, and defect personnel.

- **Determine the Importance of the Error:** Use system resources for logging only information that is important or helpful to the intended audience. Work with the hardware developer, if possible, to identify detectable errors and the information that should be relayed concerning those errors.
- **Determine the Text of the Message:** Use regular national language support (NLS) XPG/4 messages instead of the codepoints. For more information about NLS messages, see Message Facility.
- **Determine the Correct Level of Thresholding:** Each software or hardware error to be logged, can be limited by thresholding to avoid filling the error log with duplicate information. Side effects of runaway error logging include overwriting existing error log entries and unduly alarming the user. The error log is limited in size. When its size limit is reached, the log wraps. If a particular error is repeated needlessly, existing information is overwritten, which might cause inaccurate diagnostic analysis. The end user or service person can perceive a situation as more serious or pervasive than it is if they see hundreds of identical or nearly identical error entries.

  You are responsible for implementing the proper level of thresholding in the device driver code.

## Setting up Error Logging

The default size of the error log is 1 MB. As shipped, it cleans up any entries older than 30 days. To ensure that your error log entries are informative, noticed, and remain intact, *test your driver thoroughly*.

To begin error logging, do the following steps:

1. Construct error templates.
2. Add error logging calls into your code.

### *Constructing error templates*

An error template is used to associate an error with message text which is output by the `errpt` command when the error log is viewed. Error templates are described in detail in the `errupdate` command article.

You normally want to define your own error text. The error text can come from an XPG/4 message catalogue. There are also many predefined error text messages which you can view with the `errmsg -w ALL` command. For information on the use of error text in error templates, see the `errupdate` command.

### *Adding Error Logging Calls into the Code*

Typically, you define a routine in the device driver that can be called by other device driver routines when a loggable error is encountered.

The **errsave** kernel service allows the kernel and kernel extensions to write to the error log. This function takes the data passed to it, puts it into the proper structure and calls the **errsave** kernel service. The syntax for the **errsave** kernel service is:

```
#include <sys/errids.h>
void errsave(buf, cnt)
char *buf;
unsigned int cnt;
```

where:

| Item | Description |
|---|---|
| **buf** | Specifies a pointer to a buffer that contains an error record as described in the **sys/errids.h** header file. |
| **cnt** | Specifies a number of bytes in the error record contained in the buffer pointed to by the *buf* parameter. |

The following sample code is an example of a device driver error logging routine. This routine takes data passed to it from some part of the main body of the device driver. This code simply fills in the structure with the pertinent information, then passes it on using the **errsave** kernel service.

```
void
errsv_ex (int err_id, unsigned int port_num,
            int line, char *file, uint data1, uint data2)
{
    dderr   log;
    char      errbuf[255];
    ddex_dds  *p_dds;

    p_dds = dds_dir[port_num];
    log.err.error_id = err_id;

    if (port_num = BAD_STATE) {
            sprintf(log.err.resource_name, "%s :%d",
              p_dds->dds_vpd.adpt_name, data1);
            data1 = 0;
    }


 else
                sprintf(log.err.resource_name,"%s",p_dds->dds_vpd.devname);

        sprintf(errbuf, "line: %d file: %s", line, file);
```

```
        strncpy(log.file, errbuf, (size_t)sizeof(log.file));

        log.data1 = data1;
        log.data2 = data2;

        errsave(&log, (uint)sizeof(dderr));   /* run actual logging  */
}  /* end errlog_ex */
```

The data to be passed to the **errsave** kernel service is defined in the **dderr** structure, which is defined in a local header file, **dderr.h**. The definition for **dderr** is:

```
typedef struct dderr {
        struct  err_rec0 err;
        int  data1;     /* use data1 and data2 to show detail  */
        int  data2;     /* data in the errlog report. Define   */
                        /* these fields in the errlog template */
                        /* These fields may not be used in all */
                        /* cases.                              */
} dderr;
```

The first field of the **dderr.h** header file is comprised of the **err_rec0** structure, which is defined in the **sys/err_rec.h** header file. This structure contains the ID (or label) and a field for the resource name. The two data fields hold the detail data for the error log report. As an alternative, you could simply list the fields within the function.

You can also log a message into the error log from the command line. To do this, use the **errlogger** command.

Errors can be logged from applications using the `errlog` subroutine.

**Note:** Care must be taken when logging a data structure, because error logging does not support padding done by the compiler.

Consider the following data structure:

```
struct {
       struct err_rec0 err;
       long data;
} myerr;
```

Because `err_rec0` is 20 bytes in length, 0x14 bytes, the compiler normally inserts 4 bytes of padding before `data`, when compiling in 64-bit mode. The structure then looks like the following:

```
struct {
       struct err_rec0 err;
       int padding;
       long data;       /* 64 bits of data, 64-bit aligned */
} myerr;
```

Thus the `Detail_Data` item in the template begins formatting at the `padding` data item rather than at `data`.

This can be overcome, if you use the Xlc compiler as follows:

```
#pragma options align=packed
struct {
       struct err_rec0 err;
       long data;
} myerr;
#pragma options align=reset
```

After you add the templates using the **errupdate** command, compile the device driver code along with the new header file. Simulate the error and verify that it was written to the error log correctly. Some details to check for include:

- Is the error demon running? This can be verified by running the **ps -ef** command and checking for `/usr/lib/errdemon` as part of the output.

- Is the error part of the error template repository? Verify this by running the **errpt -at** command.
- Was the new header file, which was created by the **errupdate** command and which contains the error label and unique error identification number, included in the device driver code when it was compiled?

# Debug and Performance Tracing

The **trace** facility is useful for observing a running device driver and system.

The **trace** facility captures a sequential flow of time-stamped system events, providing a fine level of detail on system activity. Events are shown in time sequence and in the context of other events. The **trace** facility is useful in expanding the trace event information to understand who, when, how, and even why the event happened.

## Introduction

The operating system is shipped with permanent trace event points. These events provide general visibility to system execution. You can extend the visibility into applications by inserting additional events and providing formatting rules.

The collection of **trace** data was designed so that system performance and flow would be minimally altered by activating **trace**. Because of this, the facility is extremely useful as a performance analysis tool and as a problem determination tool.

The **trace** facility is more flexible than traditional system monitor services that access and present statistics maintained by the system. With traditional monitor services, data reduction (conversion of system events to statistics) is largely coupled to the system instrumentation. For example, the system can maintain the minimum, maximum, and average elapsed time observed for runs of a task and permit this information to be extracted.

The **trace** facility does not strongly couple data reduction to instrumentation but provides a stream of system events. It is not required to presuppose what statistics are needed. The statistics or data reduction are to a large degree separated from the instrumentation.

You can choose to develop the minimum, maximum, and average time for task A from the flow of events. But it is also possible to extract the average time for task A when called by process B, extract the average time for task A when conditions XYZ are met, develop a standard deviation for task A, or even decide that some other task, recognized by a stream of events, is more meaningful to summarize. This flexibility is invaluable for diagnosing performance or functional problems.

The **trace** facility generates large volumes of data. This data cannot be captured for extended periods of time without overflowing the storage device. This allows two practical ways that the **trace** facility can be used natively.

First, the **trace** facility can be triggered in multiple ways to capture small increments of system activity. It is practical to capture seconds to minutes of system activity in this way for post-processing. This is sufficient time to characterize major application transactions or interesting sections of a long task.

Second, the **trace** facility can be configured to direct the event stream to standard output. This allows a real time process to connect to the event stream and provide data reduction in real time, thereby creating a long term monitoring capability. A logical extension for specialized instrumentation is to direct the data stream to an auxiliary device that can either store massive amounts of data or provide dynamic data reduction.

For AIX® 5.3, tracing can be limited to a specified set of processes or threads. This can greatly reduce the amount of data generated and allow you to target the trace to report on specific tasks of interest.

You can start the system trace from:

- The command line
- SMIT
- Software

The trace facility causes predefined events to be written to a trace log. The tracing action is then stopped.

Tracing from a command line is discussed in "Controlling trace" on page 389. Tracing from a software application is discussed and an example is presented in "Examples of Coding Events and Formatting Events" on page 403.

After a trace is started and stopped, you must format it before viewing it.

To format the trace events that you have defined, you must provide a stanza that describes how the trace formatter is to interpret the data that has been collected. This is described in "Syntax for Stanzas in the trace Format File" on page 392.

The **trcrpt** command provides a general purpose report facility. The report facility provides little data reduction, but converts the raw binary event stream to a readable ASCII listing of the event stream. Data can be visually extracted by a reader, or tools can be developed to further reduce the data.

For an event to be traced, you must write an *event hook* (sometimes called a *trace hook*) into the code that you want to trace. Tracing can be done on either the system channel (channel 0) or on a generic channel (channels 1-7). All preshipped trace points are output to the system channel.

Usually, when you want to show interaction with other system routines, use the system channel. The generic channels are provided so that you can control how much data is written to the trace log. Only your data is written to one of the generic channels.

For more information on trace hooks, see "Macros for Recording trace Events" on page 390.

## Using the trace Facility

This overview explains the configuration of **trace** data collection.

### *Configuring and Starting trace Data Collection*

You can start **trace** from the command line or with a **trcstart** subroutine call. The **trcstart** subroutine is in the **librts.a** library.

The **trace** command configures the trace facility and starts data collection. The syntax of the **trcstart** subroutine is:

```
int trcstart(char *args)
```

where *args* is simply the options list desired that you would enter using the trace command if starting a system trace (channel 0). If starting a generic trace, include a **-g** option in the *args* string. On successful completion, **trcstart** returns the channel ID. For generic tracing this channel ID can be used to record to the private generic channel.

For an example of the **trcstart** routine, see "Examples of Coding Events and Formatting Events" on page 403.

When compiling a program using this subroutine, you must request the link to the **librts.a** library. Use **-l rts** as a compile option.

## Controlling trace

Basic controls for the **trace** facility exist as trace subcommands, standalone commands, and subroutines.

If you configure the **trace** routine to run asynchronously (the **-a** option), you can control the trace facility with the following commands:

| Item | Description |
|---|---|
| **trcon** | Turns collection of trace data on. |
| **trcoff** | Turns collection of trace data off. |
| **trcstop** | Stops collection of trace data (like **trcoff**) and terminates the **trace** routine. |

## Producing a trace Report

The trace report facility formats and displays the collected event stream in readable form. This report facility displays text and data for each event according to rules provided in the trace format file.

The default trace format file is **/etc/trcfmt** and contains a stanza for each event ID. The stanza for the event provides the report facility with formatting rules for that event. This technique allows you to add your own events to programs and insert corresponding event stanzas in the format file to have their new events formatted.

This report facility does not attempt to extract summary statistics (such as CPU utilization and disk utilization) from the event stream. This can be done in several other ways. To create simple summaries, consider using **awk** scripts to process the output obtained from the **trcrpt** command.

## Defining trace Events

The operating system is shipped with predefined trace hooks (events). You need only activate **trace** to capture the flow of events from the operating system.

You might want to define trace events in your code during development for tuning purposes. This provides insight into how the program is interacting with the system. The following sections provide the information that is required to do this.

### *Possible Forms of a trace Event Record*

A trace event can take several forms.

An event consists of the following:

- Hookword
- Data words (optional)
- A TID, or thread identifier
- Timestamp

An event record should be as short as possible. Many system events use only the hookword and timestamp. There is another event type you should seldom use because it is less efficient. It is a long format that allows you to record a variable length data. In this long form, the 16-bit data field of the hookword is converted to a *length* field that describes the length of the event record.

### *Macros for Recording trace Events*

The following macros should always be used to generate trace data. Do not call the tracing functions directly.

There is a macro to record each possible type of event record. The macros are defined in the **sys/trcmacros.h** header file. Most event IDs are defined in the **sys/trchkid.h** header file. Include these two header files in any program that is recording **trace** events.

The macros to record system (channel 0) events with a time stamp are:

- **TRCHKL0T** (hw)
- **TRCHKL1T** (hw,D1)
- **TRCHKL2T** (hw,D1,D2)
- **TRCHKL3T** (hw,D1,D2,D3)
- **TRCHKL4T** (hw,D1,D2,D3,D4)
- **TRCHKL5T** (hw,D1,D2,D3,D4,D5)

Similarly, to record non-time stamped system events (channel 0) on versions of AIX® prior to AIX 5L™ Version 5.3 with the 5300-05 Technology Level, use the following macros:

- **TRCHKL0** (hw)
- **TRCHKL1** (hw,D1)

- **TRCHKL2** (hw,D1,D2)
- **TRCHKL3** (hw,D1,D2,D3)
- **TRCHKL4** (hw,D1,D2,D3,D4)
- **TRCHKL5** (hw,D1,D2,D3,D4,D5)

In AIX 5L™ Version 5.3 with the 5300-05 Technology Level and above, a time stamp is recorded with each event regardless of the type of macro used.

There are only two macros to record events to one of the generic channels (channels 1-7). They are:

- **TRCGEN** (ch,hw,d1,len,buf)
- **TRCGENT** (ch,hw,d1,len,buf)

These macros record a hookword (hw), a data word (d1), and a length of data (len) specified in bytes from the user's data segment at the location specified (buf) to the event stream specified by the channel (ch). In AIX 5L™ Version 5.3 with the 5300-05 Technology Level and above, the time stamp is recorded with both macros.

### *Use of Event IDs*

Applications running on systems earlier than AIX® 6.1 and 32-bit applications running on AIX® 6.1 use 12-bit (or 3-hex-digit) event IDs (hook IDs), for a maximum of 4096 IDs. Beginning with AIX® 6.1, 64-bit applications and kernel routines can use 16-bit (or 4-hex-digit) event IDs for a maximum of 65 536 IDs. Event IDs that are permanently left in and shipped with code must be permanently assigned.

If a 16-bit ID is less than 0x1000, its least significant digit must be 0 (in the form of 0x0*hh*0 where *h* is a hexadecimal digit).

**Tip:** A 12-bit hook ID in the form of 0x*hhh* is equivalent to a 16-bit ID in the form of 0x*hhh*0 where *h* is a hexadecimal digit.

To allow you to define events in your environments or during development, a range of event IDs exist for temporary use. Event IDs for temporary use ranges from hex 010 through hex 0FF for applications running on systems earlier than AIX® 6.1 and 32-bit applications running on AIX® 6.1 and later releases. For 64-bit applications and kernel routines beginning with AIX® 6.1, event IDs for temporary use ranges from hex 0100 through hex 0FF0. No permanent (shipped) events are assigned in this range. You can freely use this range of IDs in your own environment. If you do use IDs in this range, do not let the code leave your environment.

Permanent events must have event IDs assigned by the current owner of the trace component. To obtain a trace event id, send a note with a subject of `help` to aixras@austin.ibm.com.

You should conserve event IDs because they are limited. Event IDs can be extended by the data field. The only reason to have a unique ID is that an ID is the level at which collection and report filtering is available in the trace facility. An ID can be collected or not collected by the trace collection process and reported or not reported by the trace report facility. Whole applications can be instrumented using only one event ID. The only restriction is that the granularity on choosing visibility is to choose whether events for that application are visible.

A new event can be formatted by the trace report facility (**trcrpt** command) if you create a stanza for the event in the trace format file. The trace format file is an editable ASCII file. The syntax for a format stanzas is shown in Syntax for Stanzas in the trace Format File. All permanently assigned event IDs should have an appropriate stanza in the default trace format file shipped with the base operating system.

### *Suggested Locations and Data for Permanent Events*

The intent of permanent events is to give an adequate level of visibility to determine execution, and data flow and have an adequate accounting for how CPU time is being consumed. During code development, it can be desirable to make very detailed use of trace for a component. For example, you can choose to trace the entry and exit of every subroutine in order to understand and tune path length. However, this would generally be an excessive level of instrumentation to ship for a component.

Consult a performance analyst for decisions regarding what events and data to capture as permanent events for a new component. The following paragraphs provide some guidelines for these decisions.

Events should capture execution flow and data flow between major components or major sections of a component. For example, there are existing events that capture the interface between the virtual memory manager and the logical volume manager. If work is being queued, data that identifies the queued item (a handle) should be recorded with the event. When a queue element is being processed, the "dequeue" event should provide this identifier as data also, so that the queue element being serviced is identified.

Data or requests that are identified by different handles at different levels of the system should have events and data that allow them to be uniquely identified at any level. For example, a read request to the physical file system is identified by a file descriptor and a current offset in the file. To a virtual memory manager, the same request is identified by a segment ID and a virtual page address. At the disk device driver level, this request is identified as a pointer to a structure that contains pertinent data for the request.

The file descriptor or segment information is not available at the device driver level. Events must provide the necessary data to link these identifiers so that, for example, when a disk interrupt occurs for incoming data, the identifier at that level (which can simply be the buffer address for where the data is to be copied) can be linked to the original user request for data at some offset into a file.

Events should provide visibility to major protocol events such as requests, responses, acknowledgements, errors, and retries. If a request at some level is fragmented into multiple requests, a trace event should indicate this and supply linkage data to allow the multiple requests to be tracked from that point. If multiple requests at some level are coalesced into a single request, a trace event should also indicate this and provide appropriate data to track the new request.

Use events to give visibility to resource consumption. Whenever resources are claimed, returned, created, or deleted an event should record the fact. For example, claiming or returning buffers to a buffer pool or growing or shrinking the number of buffers in the pool.

The following guidelines can help you determine where and when you should have trace hooks in your code:

- Tracing entry and exit points of every function is not necessary. Provide only key actions and data.
- Show linkage between major code blocks or processes.
- If work is queued, associate a name (handle) with it and output it as data.
- If a queue is being serviced, the trace event should indicate the unique element being serviced.
- If a work request or response is being referenced by different handles as it passes through different software components, trace the transactions so the action or receipt can be identified.
- Place trace hooks so that requests, responses, errors, and retries can be observed.
- Identify when resources are claimed, returned, created, or destroyed.

Also note that:

- A trace ID can be used for a group of events by "turning" on one of the data fields. This means that a particular data field can be used to identify from where the trace point was called. The trace format routine can be made to format the trace data for that unique trace point.
- The trace hook is the level at which a group of events can be enabled or disabled. Note that trace hooks can be grouped in SMIT. For more information, see “Trace Event Groups” on page 406.

### *Syntax for Stanzas in the trace Format File*

The intent of the trace format file is to provide rules for presentation and display of the expected data for each event ID. This allows new events to be formatted without changing the report facility. Rules for new events are simply added to the format file. The syntax of the rules provide flexibility in the presentation of the data.

A trace format stanza can be as long as required to describe the rules for any particular event. The stanza can be continued to the next line by terminating the present line with a backslash (\). The fields are:

**event_id**
Each stanza begins with the 3-digit hexadecimal event ID that the stanza describes, followed by a space.

**Note:** Beginning with AIX® 6.1, each stanza can also begin with a 4-digit hexadecimal event ID.

**V.R**
This field describes the version (V) and release (R) that the event was first assigned. Any integers work for V and R, and you might want to keep your own tracking mechanism.

**L=**
The text description of an event can begin at various indentation levels. This improves the readability of the report output. The indentation levels correspond to the level at which the system is running. The recognized levels are:

**APPL**
Application level

**SVC**
Transitioning system call

**KERN**
Kernel level

**INT**
Interrupt

**event_label**
The *event_label* is an ASCII text string that describes the overall use of the event ID. This is used by the **-j** option of the **trcrpt** command to provide a listing of events and their first level description. The event label also appears in the formatted output for the event unless the `event_label` field starts with an **@** character.

**\n**
The event stanza describes how to parse, label, and present the data contained in an event record. You can insert a **\n** (newline) in the event stanza to continue data presentation of the data on a new line. This allows the presentation of the data for an event to be several lines long.

**\t**
The **\t** (tab) function inserts a tab at the point it is encountered in parsing the description. This is similar to the way the **\n** function inserts new lines. Spacing can also be inserted by spaces in the `data_label` or `match_label` fields.

**starttimer(#,#)**
The `starttimer` and `endtimer` fields work together. The (#,#) field is a unique identifier that associates a particular `starttimer` value with an `endtimer` that has the same identifier. By convention, if possible, the identifiers should be the ID of starting event and the ID of the ending event.

When the report facility encounters a start timer directive while parsing an event, it associates the starting events time with the unique identifier. When an end timer with the same identifier is encountered, the report facility outputs the delta time (this appears in brackets) that elapsed between the starting event and ending event. The begin and end system call events make use of this capability. On the return from system call event, a delta time is shown that indicates how long the system call took.

**endtimer(#,#)**
See the `starttimer` field in the preceding paragraph.

**data_descriptor**
The `data_descriptor` field is the fundamental field that describes how the report facility consumes, labels, and presents the data.

The various subfields of the `data_descriptor` field are:

**data_label**
The data label is an ASCII string that can optionally precede the output of data consumed by the following `format` field.

**format**
You can think of the report facility as having a pointer into the data portion of an event. This data pointer is initialized to point to the beginning of the event data (the 16-bit data field in the hookword). The format field describes how much data the report facility consumes from this point and how the data is considered. For example, a value of **Bm.n** tells the report facility to consume m bytes and n bits of data and to consider it as binary data.

The possible format fields are described in the following section. If this field is not followed by a comma, the report facility outputs the consumed data in the format specified. If this field is followed by a comma, it signifies that the data is not to be displayed but instead compared against the following `match_vals` field. The data descriptor associated with the matching `match_val` field is then applied to the remainder of the data.

**match_val**
The match value is data of the same format described by the preceding format fields. Several match values typically follow a format field that is being matched. The successive match fields are separated by commas. The last match value is not followed by a comma. Use the character string * as a pattern-matching character to match anything. A pattern-matching character is frequently used as the last element of the `match_val` field to specify default rules if the preceding `match_val` field did not occur.

**match_label**
The match label is an ASCII string that is output for the corresponding match.

Each of the possible `format` fields is described in the comments of the **/etc/trcfmt** file. The following shows several possibilities:

| Item | Description |
|---|---|
| **Format field** | descriptions |

| Item | Description |
|---|---|
| In most cases, the data length part of the specifier can also be the letter "**W**" which indicates that the word size of the trace hook is to be used. For example, **XW** will format 4 or 8 bytes into hexadecimal, depending upon whether the trace hook comes from a 32 or 64 bit environment. | |
| **Am.n** | This value specifies that m bytes of data are consumed as ASCII text, and that it is displayed in an output field that is *n* characters wide. The data pointer is moved *m* bytes. |
| **S1, S2, S4** | Left justified string. The length of the field is defined as 1 byte (S1), 2 bytes (S2), or 4 bytes (S4) and so on. The data pointer is moved accordingly. **SW** indicates that the word size for the trace event is to be used. |
| **Bm.n** | Binary data of *m* bytes and *n* bits. The data pointer is moved accordingly. |
| **Xm** | Hexadecimal data of *m* bytes. The data pointer is moved accordingly. |

| Item | Description |
|---|---|
| **D2, D4** | Signed decimal format. Data length of 2 (D2) bytes or 4 (D4) bytes is consumed. |
| **U2, U4** | Unsigned decimal format. Data length of 2 or 4 bytes is consumed. |
| **F4, F8** | Floating point of 4 or 8 bytes. |
| **Gm.n** | Positions the data pointer. It specifies that the data pointer is positioned *m* bytes and *n* bits into the data. |
| **Om.n** | Skip or omit data. It omits *m* bytes and *n* bits. |
| **Rm** | Reverse the data pointer *m* bytes. |
| **Wm** | Position DATA_POINTER at word *m*. The word size is either 4 or 8 bytes, depending upon whether or not this is a 32 or 64 bit format trace. This bares no relation to the %W format specifier. |
| **m8** | Output the next 8 bytes as time in milliseconds from the beginning of the trace. **mW** will format only 8 bytes of data. The DATA_POINTER is advanced by 8 bytes. |
| **u4**, **u8** | Output the next 4 or 8 bytes as time in microseconds. **mW** will format either 4 or 8 bytes of data depending on whether the current hook is 32 or 64 bits. The DATA_POINTER is advanced by 4 or 8 bytes. |

Some macros are provided that can be used as format fields to quickly access data. For example:

| Item | Description |
|---|---|
| $D1, $D2, $D3, $D4, $D5 | These macros access data words 1 through 5 of the event record without moving the data pointer. The data accessed by a macro is hexadecimal by default. A macro can be cast to a different data type (X, D, U, B) by using a % character followed by the new format code. For example, the following macro causes data word one to be accessed, but to be considered as 2 bytes and 3 bits of binary data:<br>`$D1%B2.3` |
| $HD | This macro accesses the first 16 bits of data contained in the hookword, in a similar manner as the $D1 through $D5 macros access the various data words. It is also considered as hexadecimal data, and also can be cast. |

You can define other macros and use other formatting techniques in the trace format file. This is shown in the following trace format file example.

### *Example Trace Format File*

You can define other macros and use other formatting techniques in the trace format file.

```
# @(#)65  1.142  src/bos/usr/bin/trcrpt/trcfmt, cmdtrace, bos43N, 9909A_43N 2/12/99 13:15:34
# COMPONENT_NAME: CMDTRACE   system trace logging and reporting facility
#
# FUNCTIONS: template file for trcrpt
#
# ORIGINS: 27, 83
#
# (C) COPYRIGHT International Business Machines Corp. 1988, 1993
# All Rights Reserved
# Licensed Materials - Property of IBM
#
# US Government Users Restricted Rights - Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
#
```

```
# LEVEL 1, 5 Years Bull Confidential Information
#

# I. General Information
#
#    The formats shown below apply to the data placed into the
#    trcrpt format buffer.  These formats in general mirror the binary
#    format of the data in the trace stream.  The exceptions are
#    hooks from a 32-bit application on a 64-bit kernel, and hooks from a
#    64-bit application on a 32-bit kernel.  These exceptions are noted
#    below as appropriate.
#
#    Trace formatting templates should not use the thread id or time
#    stamp from the buffer.  The thread id should be obtained with the
#    $TID macro.  The time stamp is a raw timer value used by trcrpt to
#    calculate the elapsed and delta times.  These values are either
#    4 or 8 bytes depending upon the system the trace was run on, not upon
#    the environment from which the hook was generated.
#    The system environment, 32 or 64 bit, and the hook's
#    environment, 32 or 64 bit, are obtained from the $TRACEENV and $HOOKENV
#    macros discussed below.
#
#    To interpret the time stamp, it is necessary to get the values from
#    hook 0x00a, subhook 0x25c, used to convert it to nanoseconds.
#    The 3 data words of interest are all 8 bytes in length and are in
#    the generic buffer, see the template for hook 00A.
#    The first data word gives the multiplier, m, and the second word
#    is the divisor, d.  These values should be set to 1 if the
#    third word doesn't contain a 2.  The nanosecond time is then
#    calculated with nt = t * m / d where t is the time from the trace.
#
#    Also, on a 64-bit system, there will be a header on the trace stream.
#    This header serves to identify the stream as coming from a
#    64-bit system.  There is no such header on the data stream on a
#    32-bit system.  This data stream, on both systems, is produced with
#    the "-o -" option of the trace command.
#    This header consists only of a 4-byte magic number, 0xEFDF1114.
#
# A. Binary format for the 32-bit trace data
#    TRCHKL0        MMMTDDDDiiiiiiii
#    TRCHKL0T       MMMTDDDDiiiiiiiittttttttt
#    TRCHKL1        MMMTDDDD11111111iiiiiiii
#    TRCHKL1T       MMMTDDDD11111111iiiiiiiittttttttt
#    Note that trchkg covers TRCHKL2-TRCHKL5.
#    trchkg       MMMTDDDD11111111222222223333333344444444555555551iiiiiiii
#    trchkgt      MMMTDDDD11111111222222223333333344444444555555551 i... t...
#    trcgent      MMMTLLLL11111111vvvvvvvvvvvvvvvvvvvvvvvvvvvvxxxxxx i... t...
#
#        legend:
#      MMM = hook id
#      T   = hooktype
#      D   = hookdata
#      i   = thread id, 4 bytes on a 32 byte system and 8 bytes on a 64-bit
#            system. The thread id starts on a 4 or 8 byte boundary.
#      t   = timestamp, 4 bytes on a 32-bit system or 8 on a
#            64-bit system.
#      1   = d1  (see trchkid.h for calling syntax for the tracehook routines)
#      2   = d2, etc.
#      v   = trcgen variable length buffer
#      L   = length of variable length data in bytes.
#
#    The DATA_POINTER starts at the third byte in the event, ie.,
#    at the 16 bit hookdata DDDD.
#    The trcgen() is an exception. The DATA_POINTER starts at
#    the fifth byte, ie., at the 'd1' parameter 11111111.
#
#    Note that a generic trace hook with a hookid of 0x00b is
#    produced for 64-bit data traced from a 64-bit app running on
#    a 32-bit kernel.  Since this is produced on a 32-bit system, the
#    thread id and time stamp will be 4 bytes in the data stream.
#
# B.  64-bit trace hook format
#
#    TRCHK64L0    ffffllllhhhhssss iiiiiiiiiiiiiiii
#    TRCHK64L0T    ffffllllhhhhssss iiiiiiiiiiiiiiii ttttttttttttttttt
#    TRCHK64L1    ffffllllhhhhssss 1111111111111111 i...
#    ...
#    TRCGEN       ffffllllhhhhssss dddddddddddddddd "string" i...
#    TRCGENt      ffffllllhhhhssss dddddddddddddddd "string" i... t...
#
#    Legend
#      f - flags
```

```
#        tgbuuuuuuuuuuuuu: t - time stamped,  g - generic (trcgen),
#          b - 32-bit data, u - unused.
#      l - length, number of bytes traced.
#        For TRCHKL0 llll = 0,
#        for TRCHKL5T llll = 40, 0x28 (5 8-byte words)
#      h - hook id
#       s - subhook id
#       1 - data word 1, ...
#       d - generic trace data word.
#       i - thread id, 8 bytes on a 64-bit system, 4 on a 32-bit system.
#           The thread id starts on an 8-byte boundary.
#       t - time stamp, 8 bytes on a 64-bit system, 4 on a 32-bit system.
#
#    For non-generic entries, the data pointer starts at the
#    subhook id, offset 6.  This is compatible with the 32-bit
#    hook format shown above.
#    For generic (trcgen) hooks, the g flag above is on.  The
#    length shows the number of variable bytes traced and does not include
#    the data word.
#    The data pointer starts at the 64-bit data word.
#    Note that the data word is 64 bits here.
#
# C. Trace environments
#    The trcrpt, trace report, utility must be able to tell whether
#    the trace it's formatting came from a 32 or a 64 bit system.
#    This is accomplished by the log file header's magic number.
#    In addition, we need to know whether 32 or 64 bit data was traced.
#    It is possible to run a 32-bit application on a 64-bit kernel,
#    and a 64-bit application on a 32-bit kernel.
#    In the case of a 32-bit app on a 64-bit kernel, the "b" flag
#    shown under item B above is set on.  The trcrpt program will
#    then present the data as if it came from a 32-bit kernel.
#    In the second case, if the reserved hook id 00b is seen, the data
#    traced by the 32-bit kernel is made to look as if it came
#    from a 64-bit trace.  Thus the templates need not be kernel aware.
#
#    For example, if a 32-bit app uses
#    TRCHKL5T(0x50000005, 1, 2, 3, 4, 5)
#    and is running on a 64-bit kernel, the data actually traced
#    will look like:
#      ffffllllhhhhssss 1111111111111111 2222222222222222 3333333333333333
#      a000001450000005 0000000100000002 0000000300000004 0000000500000000 i t
#    Here, the flags have the T and B bits set (a000) which says
#    the hook is timestamped and from a 32-bit app.
#    The length is 0x14 bytes, 5 4-byte registers 00000001 through
#    00000005.
#    The hook id is 0x5000.
#    The subhook id is 0x0005.
#    i and t refer to the 8-byte thread id and time stamp.
#
#    This would be reformatted as follows before being processed
#    by the corresponding template:
#      500e0005 00000001 00000002 00000003 00000004 00000005
#    Note this is how the data would look if traced on a 32-bit kernel.
#    Note also that the data would be followed by an 8-byte thread id and
#    time stamp.
#
#    Similarly, consider the following hook traced by a 64-bit app
#    on a 32-bit kernel:
#      TRCHKL5T(0x50000005, 1, 2, 3, 4, 5)
#    The data traced would be:
#      00b8002c 80000028 50000005 0000000000000001 ... 0000000000000005 i t
#    Note that this is a generic trace entry, T = 8.
#    In the generic entry, we're using the 32-bit data word for the flags
#    and length.
#    The trcrpt utility would reformat this before processing by
#    the template as follows:
#      8000002850000005 0000000000000001 ... 0000000000000005 i8 t8
#
#    The thread id and time stamp in the data stream will be 4 bytes,
#    because the hook came from a 32-bit system.
#
#    If a 32-bit app traces generic data on a 64-bit kernel, the b
#    bit will be set on in the data stream, and the entry will be formatted
#    like it came from a 32-bit environment, i.e. with a 32-bit data word.
#    For the case of a 64-bit app on a 32-bit kernel, generic trace
#    data is handled in the same manner, with the flags placed
#    into the data word.
#    For example, if the app issues
#      TRCGEN(1, 0x50000005, 1, 6, "hello")
#    The 32-bit kernel trace will generate
#      00b00012 40000006 50000005 0000000000000001 "hello"
```

```
#   This will be reformatted by trcrpt into
#      4000000650000005 0000000000000001 "hello"
#   with the data pointer starting at the data word.
#
#   Note that the string "hello" could have been 4096 bytes.  Therefore
#   this generic entry must be able to violate the 4096 byte length
#   restriction.
#
# D. Indentation levels
#    The left margin is set per template using the 'L=XXXX' command.
#    The default is L=KERN, the second column.
#    L=APPL moves the left margin to the first  column.
#    L=SVC  moves the left margin to the second column.
#    L=KERN moves the left margin to the third  column.
#    L=INT  moves the left margin to the fourth column.
#    The command if used must go just after the version code.
#
#    Example usage:
#113 1.7 L=INT "stray interrupt" ... \
#
# E. Continuation code and delimiters.
#    A '\' at the end of the line must be used to continue the template
#      on the next line.
#    Individual strings (labels) can be separated by one or more blanks
#      or tabs. However, all whitespace is squeezed down to 1 blank on
#      the report. Use '\t' for skipping to the next tabstop, or use
#      A0.X format (see below) for variable space.
#
#
# II. FORMAT codes
#
# A. Codes that manipulate the DATA_POINTER
# Gm.n
#     "Goto"    Set DATA_POINTER to byte.bit location m.n
#
# Om.n
#     "Omit"    Advance DATA_POINTER by m.n byte.bits
#
# Rm
#     "Reverse" Decrement DATA_POINTER by m bytes. R0 byte aligns.
#
# Wm
#     Position DATA_POINTER at word m.  The word size is either 4 or 8
#     bytes, depending upon whether or not this is a 32 or 64 bit format
#     trace.  This bares no relation to the %W format specifier.
#
# B. Codes that cause data to be output.
# Am.n
#     Left justified ascii.
#     m=length in bytes of the binary data.
#     n=width of the displayed field.
#     The data pointer is rounded up to the next byte boundary.
#     Example
#      DATA_POINTER|
#                  V
#             xxxxxhello world\0xxxxxx
#
#  i.   A8.16 results in:                       |hello wo        |
#       DATA_POINTER--------|
#                           V
#              xxxxxhello world\0xxxxxx
#
#  ii.  A16.16 results in:                      |hello world     |
#       DATA_POINTER----------------|
#                                   V
#              xxxxxhello world\0xxxxxx
#
#  iii. A16 results in:                         |hello world|
#       DATA_POINTER----------------|
#                                   V
#              xxxxxhello world\0xxxxxx
#
#  iv.  A0.16 results in:                       |               |
#       DATA_POINTER|
#                   V
#              xxxxxhello world\0xxxxxx
#
# Sm (m = 1, 2, 4, or 8)
#     Left justified ascii string.
#     The length of the string is in the first m bytes of
#      the data. This length of the string does not include these bytes.
#     The data pointer is advanced by the length value.
```

```
#     SW specifies the length to be 4 or 8 bytes, depending upon whether
#     this is a 32 or 64 bit hook.
#     Example
#      DATA_POINTER|
#                  V
#             xxxxxBhello worldxxxxxx     (B = hex 0x0b)
#
#  i.   S1 results in:                          |hello world|
#       DATA_POINTER-----------|
#                              V
#               xxxxBhello worldxxxxxx
#
# $reg%S1
#     A register with the format code of 'Sx' works "backwards" from
#     a register with a different type. The format is Sx, but the length
#     of the string comes from $reg instead of the next n bytes.
#
# Bm.n
#     Binary format.
#     m = length in bytes
#     n = length in bits
#     The length in bits of the data is m * 8 + n. B2.3 and B0.19 are the same.
#     Unlike the other printing FORMAT codes, the DATA_POINTER
#     can be bit aligned and is not rounded up to the next byte boundary.
#
# Xm
#     Hex format.
#     m = length in bytes. m=0 thru 16
#     X0 is the same as X1, except that no trailing space is output after
#     the data.  Therefore X0 can be used with a LOOP to output an
#     unbroken string of data.
#     The DATA_POINTER is advanced by m (1 if m = 0).
#     XW will format either 4 or 8 bytes of data depending upon whether
#     this is a 32 or 64 bit hook.  The DATA_POINTER is advanced
#     by 4 or 8 bytes.
#
# Dm (m = 2, 4, or 8)
#     Signed decimal format.
#     The length of the data is m bytes.
#     The DATA_POINTER is advanced by m.
#     DW will format either 4 or 8 bytes of data depending upon whether
#     this is a 32 or 64 bit hook.  The DATA_POINTER is advanced
#     by 4 or 8 bytes.
#
# Um (m = 2, 4, or 8)
#     Unsigned decimal format.
#     The length of the data is m bytes.
#     The DATA_POINTER is advanced by m.
#     UW will format either 4 or 8 bytes of data depending upon whether
#     this is a 32 or 64 bit hook.  The DATA_POINTER is advanced
#     by 4 or 8 bytes.
#
# om (m = 2, 4, or 8)
#     Octal format.
#     The length of the data is m bytes.
#     The DATA_POINTER is advanced by m.
#     ow will format either 4 or 8 bytes of data depending upon whether
#     this is a 32 or 64 bit hook.  The DATA_POINTER is advanced
#     by 4 or 8 bytes.
#
# F4
#     Floating point format. (like %0.4E)
#     The length of the data is 4 bytes.
#     The format of the data is that of C type 'float'.
#     The DATA_POINTER is advanced by 4.
#
# F8
#     Floating point format. (like %0.4E)
#     The length of the data is 8 bytes.
#     The format of the data is that of C type 'double'.
#     The DATA_POINTER is advanced by 8.
#
# HB
#     Number of bytes in trcgen() variable length buffer.
#     The DATA_POINTER is not changed.
#
# HT
#   32-bit hooks:
#     The hooktype. (0 - E)
#     trcgen  = 0, trchk  = 1, trchl  = 2, trchkg  = 6
#     trcgent = 8, trchkt = 9, trchlt = A, trchkgt = E
#     HT & 0x07 masks off the timestamp bit
```

```
#     This is used for allowing multiple, different trchook() calls with
#       the same template.
#     The DATA_POINTER is not changed.
#   64-bit hooks
#     This is the flags field.
#     0x8000 - hook is time stamped.
#     0x4000 - This is a generic trace.
#
#     Note that if the hook was reformatted as discussed under item
#     I.C above, HT is set to reflect the flags in the new format.
#
# C. Codes that interpret the data in some way before output.
# Tm (m = 4, or 8)
#     Output the next m bytes as a data and time string,
#     in GMT timezone format.  (as in ctime(&seconds))
#     The DATA_POINTER is advanced by m bytes.
#     Only the low-order 32-bits of the time are actually used.
#     TW will format either 4 or 8 bytes of data depending upon whether
#     this is a 32 or 64 bit hook.  The DATA_POINTER is advanced
#     by 4 or 8 bytes.
#
# Em (m = 1, 2, 4, or 8)
#     Output the next m bytes as an 'errno' value, replacing
#     the numeric code with the corresponding #define name in
#     /usr/include/sys/errno.h
#     The DATA_POINTER is advanced by 1, 2, 4, or 8.
#     EW will format either 4 or 8 bytes of data depending upon whether
#     this is a 32 or 64 bit hook.  The DATA_POINTER is advanced
#     by 4 or 8 bytes.
#
# Pm (m = 4, or 8)
#     Use the next m bytes as a process id (pid), and
#     output the pathname of the executable with that process id.
#     Process ids and their pathnames are acquired by the trace command
#     at the start of a trace and by trcrpt via a special EXEC tracehook.
#     The DATA_POINTER is advanced by 4 or 8 bytes.
#     PW will format either 4 or 8 bytes of data depending upon whether
#     this is a 32 or 64 bit hook.
#
# \t
#     Output a tab. \t\t\t outputs 3 tabs. Tabs are expanded to spaces,
#     using a fixed tabstop separation of 8. If L=0 indentation is used,
#     the first tabstop is at 3.
#
# \n
#     Output a newline. \n\n\n outputs 3 newlines.
#     The newline is left-justified according to the INDENTATION LEVEL.
#
# $macro
#     Undefined macros have the value of 0.
#     The DATA_POINTER is not changed.
#     An optional format can be used with macros:
#        $v1%X8    will output the value $v1 in X8 format.
#        $zz%B0.8  will output the value $v1 in 8 bits of binary.
#     Understood formats are: X, D, U, B and W. Others default to X2.
#
#     The W format is used to mask the register.
#     Wm.n masks off all bits except bits m through n, then shifts the
#     result right m bits.  For example, if $ZZ = 0x12345678, then
#     $zz%W24.27 yields 2.  Note the bit numbering starts at the right,
#     with 0 being the least significant bit.
#
# "string"     'string' data type
#     Output the characters inside the double quotes exactly. A string
#     is treated as a descriptor. Use "" as a NULL string.
#
# `string format $macro` If a string is backquoted, it is expanded
#     as a quoted string, except that FORMAT codes and $registers are
#     expanded as registers.
#
# III. SWITCH statement
#     A format code followed by a comma is a SWITCH statement.
#     Each CASE entry of the SWITCH statement consists of
#       1. a 'matchvalue' with a type (usually numeric) corresponding to
#          the format code.
#       2. a simple 'string' or a new 'descriptor' bounded by braces.
#          A descriptor is a sequence of format codes, strings, switches,
#          and loops.
#       3. and a comma delimiter.
#       The switch is terminated by a CASE entry without a comma delimiter.
#     The CASE entry selected is the first entry whose matchvalue
#     is equal to the expansion of the format code.
```

```
#     The special matchvalue '\*' is a wildcard and matches anything.
#     The DATA_POINTER is advanced by the format code.
#
#
# IV. LOOP statement
#     The syntax of a 'loop' is
#     LOOP format_code { descriptor }
#     The descriptor is executed N times, where N is the numeric value
#       of the format code.
#     The DATA_POINTER is advanced by the format code plus whatever the
#       descriptor does.
#     Loops are used to output binary buffers of data, so descriptor is
#       usually simply X1 or X0. Note that X0 is like X1 but does not
#       supply a space separator ' ' between each byte.
#
#
# V. macro assignment and expressions
#   'macros' are temporary (for the duration of that event) variables
#   that work like shell variables.
#   They are assigned a value with the syntax:
#   {{ $xxx = EXPR }}
#   where EXPR is a combination of format codes, macros, and constants.
#   Allowed operators are + - / *
#   For example:
#{{ $dog = 7 + 6 }} {{ $cat = $dog * 2 }} $dog $cat
#
#   will output:
#000D 001A
#
#   Macros are useful in loops where the loop count is not always
#   just before the data:
#G1.5 {{ $count = B0.5 }} G11 LOOP $count {X0}
#
#   Up to 255 macros can be defined per template.
#
#
# VI. Special macros:
# $HOOKENV     This is either "32" or "64" depending upon
#              whether this is a 32 or 64 bit trace hook.
#              This can be used to interpret the HT value.
# $TRACEENV    This is either "32" or "64" depending upon
#              whether this is a 32 or 64 bit trace, i.e., whether the
#
         trace was generated by a 32 or 64 bit kernel.
#              Since hooks will be formatted according to the environment
#              they came from, $HOOKENV should normally be used.
# $RELLINENO   line number for this event. The first line starts at 1.
# $D1 - $D5    dataword 1 through dataword 5. No change to datapointer.
#   The data word is either 4 or 8 bytes.
# $L1 - $L5    Long dataword 1,5(64 bits). No change to datapointer.
# $HD          hookdata (lower 16 bits)
#              For a 32-bit generic hook, $HD is the length of the
#              generic data traced.
#              For 32 or 64 bit generic hooks, use $HL.
# $HL          Hook data length.  This is the length in bytes of the hook
#              data.  For generic entries it is the length of the
#              variable length buffer and doesn't include the data word.
# $WORDSIZE    Contains the word size, 4 or 8 bytes, of the current
#              entry, (i.e.) $HOOKENV / 8.
# $GENERIC     specifies whether the entry is a generic entry.  The
#              value is 1 for a generic entry, and 0 if not generic.
#              $GENERIC is especially useful if the hook can come from
#              either a 32 or 64 bit environment, since the types (HT)
#              have different formats.
# $TOTALCPUS   Output the number of CPUs in the system.
# $TRACEDCPUS  Output the number of CPUs that were traced.
# $REPORTEDCPUS Output the number of CPUs active in this report.
#              This can decrease as CPUs stop tracing when, for example,
#              the single-buffer trace, -f, was used and the buffers for
#              each CPU fill up.
# $LARGEDATATYPES This is set to 1 if the kernel is supporting large data
#              types for 64-bit applications.
# $SVC         Output the name of the current SVC
# $EXECPATH    Output the pathname of the executable for current process.
# $PID         Output the current process id.
# $TID         Output the current thread id.
# $CPUID       Output the current processor id.
# $PRI         Output the current process priority
# $ERROR       Output an error message to the report and exit from the
#              template after the current descriptor is processed.
#              The error message supplies the logfile, logfile offset of the
#              start of that event, and the traceid.
```

```
# $LOGIDX      Current logfile offset into this event.
# $LOGIDX0     Like $LOGIDX, but is the start of the event.
# $LOGFILE     Name of the logfile being processed.
# $TRACEID     Traceid of this event.
# $DEFAULT     Use the DEFAULT template 008
# $STOP        End the trace report right away
# $BREAK       End the current trace event
# $SKIP        Like break, but don't print anything out.
# $DATAPOINTER The DATA_POINTER. It can be set and manipulated
#              like other user-macros.
#              {{ $DATAPOINTER = 5 }} is equivalent to G5
#
# Note:  For generic trace hooks, $DATAPOINTER points to the
#   data word.  This means it is 0x4 for 32-bit hooks, and 0x8 for
#   64-bit hooks.
#   For non-generic hooks, $DATAPOINTER is set to 2 for 32-bit hooks
#   and to 6 for 64 bit trace hooks.  This means it always
#   points to the subhook id.
#
# $BASEPOINTER Usually 0. It is the starting offset into an event. The actual
#              offset is the DATA_POINTER + BASE_POINTER. It is used with
#              template subroutines, where the parts on an event have the
#              same structure, and can be printed by the same template, but
#              might have different starting points into an event.
# $IPADDR      IP address of this machine, 4 bytes.
# $BUFF        Buffer allocation scheme used, 1=kernel heap, 2=separate segment.
#
# VII. Template subroutines
#    If a macro name consists of 3 hex digits, it is a "template subroutine".
#    The template whose traceid equals the macro name is inserted in place
#    of the macro.
#
#    The data pointer is where it was when the template
#    substitution was encountered. Any change made to the data pointer
#    by the template subroutine remains in affect when the template ends.
#
#    Macros used within the template subroutine correspond to those in the
#    calling template. The first definition of a macro in the called template
#    is the same variable as the first in the called. The names are not
#    related.
#
#    NOTE: Nesting of template subroutines is supported to 10 levels.
#
#    Example:
#    Output the trace label ESDI STRATEGY.
#    The macro '$stat' is set to bytes 2 and 3 of the trace event.
#    Then call template 90F to interpret a buf header. The macro '$return'
#    corresponds to the macro '$rv', because they were declared in the same
#    order. A macro definition with no '=' assignment just declares the name
#    like a place holder. When the template returns, the saved special
#    status word is output and the returned minor device number.
#
#900 1.0 "ESDI STRATEGY" {{ $rv = 0 }} {{ $stat = X2 }} \
#       $90F \n\
#special_esdi_status=$stat for minor device $rv
#
#90F 1.0 "" G4 {{ $return }} \
#       block number X4 \n\
#       byte count   X4 \n\
#       B0.1, 1 B_FLAG0 \
#       B0.1, 1 B_FLAG1 \
#       B0.1, 1 B_FLAG2 \
#       G16 {{ $return = X2 }}
#
#
#    Note: The $DEFAULT reserved macro is the same as $008
#
# VIII. BITFLAGS statement
#     The syntax of a 'bitflags' is
#     BITFLAGS [format_code|register],
#         flag_value string {optional string if false},   or
#         '&' mask field_value string,
#                 ...
#
#     This statement simplifies expanding state flags, because it looks
#       a lot like a series of #defines.
#     The '&' mask is used for interpreting bit fields.
#     The mask is anded to the register and the result is compared to
#       the field_value. If a match, the string is printed.
#     The base is 16 for flag_values and masks.
#     The DATA_POINTER is advanced if a format code is used.
```

```
#     Note: the default base for BITFLAGS is 16. If the mask or field value
#     has a leading "o", the number is octal. 0x or 0X makes the number hexadecimal.
```

## *Examples of Coding Events and Formatting Events*

There are five basic steps involved in generating a trace from your software program.

Those steps are as follows:

*Step 1: Enable the trace*
Enable and disable the trace from your software that has the trace hooks defined.

The following code shows the use of trace events to time the running of a program loop.

```
#include      <sys/trcctl.h>
#include      <sys/trcmacros.h>
#include      <sys/trchkid.h>

char      *ctl_file = "/dev/systrctl";
int      ctlfd;
int      i;

main()
{
        printf("configuring trace collection \n");
        if (trcstart("-ad")){
                perror("trcstart");
                exit(1);
        }
        printf("turning trace on \n");
        if(trcon(0)){
                perror("TRCON");
                exit(1);
        }
        /* here is the code that is being traced */
        for(i=1;i<11;i++){
                TRCHKL1T(HKWD_USER1,i);
                /*   sleep(1) */
                /* you can uncomment sleep to make the loop
                /* take longer. If you do, you will want to
                /* filter the output or you will be */
                /* overwhelmed with 11 seconds of data */
        }
        /* stop tracing code */
        printf("turning trace off\n");
        if(trcstop(0)){
                perror("TRCOFF");
                exit(1);
        }
```

*Step 2: Compile your program*
This next step is to compile your program.

When you compile the sample program, you need to link to the **librts.a** library:

```
cc -o sample sample.c -l rts
```

*Step 3: Run the program*
Run the program.

In this case, it can be done with the following command:

```
./sample
```

*Step 4: Add a stanza to the format file*
This provides the report generator with the information to correctly format your file. The report facility does not know how to format the **HKWD_USER1** event, unless you provide rules in the trace format file.

The following is an example of a stanza for the **HKWD_USER1** event. The **HKWD_USER1** event is event ID 010 hexadecimal. You can verify this by looking at the **sys/trchkid.h** header file.

```
# User event HKWD_USER1 Formatting Rules Stanza
# An example that will format the event usage of the sample program
010 1.0 L=APPL "USER EVENT - HKWD_USER1" O2.0 \n\
               "The # of loop iterations =" U4\n\
               "The elapsed time of the last loop = "\
                endtimer(0x010,0x010) starttimer(0x010,0x010)
```

**Note:** When entering the example stanza, do not modify the master format file **/etc/trcfmt**. Instead, make a copy and keep it in your own directory. This allows you to always have the original trace format file available. If you are going to ship your formatting stanzas, the **trcupdate** command is used to add your stanzas to the default trace format file.

*Step 5: Run the format/filter program*
Filter the output report to get only your events.

To do this, run the **trcrpt** command:

```
trcrpt -d 010 -t mytrcfmt -O exec=on -o sample.rpt
```

The formatted trace results are:

```
ID PROC NAME  I   ELAPSED_SEC  DELTA_MSEC    APPL     SYSCALL    KERNEL   INTERRUPT
010   sample     0.000105984  0.105984     USER HOOK 1
                                           The data field for the user hook = 1
010   sample     0.000113920  0.007936     USER HOOK 1
                                           The data field for the user hook = 2 [7 usec]
010   sample     0.000119296  0.005376     USER HOOK 1
                                           The data field for the user hook = 3 [5 usec]
010   sample     0.000124672  0.005376     USER HOOK 1
                                           The data field for the user hook = 4 [5 usec]
010   sample     0.000129792  0.005120     USER HOOK 1
                                           The data field for the user hook = 5 [5 usec]
010   sample     0.000135168  0.005376     USER HOOK 1
                                           The data field for the user hook = 6 [5 usec]
010   sample     0.000140288  0.005120     USER HOOK 1
                                           The data field for the user hook = 7 [5 usec]
010   sample     0.000145408  0.005120     USER HOOK 1
                                           The data field for the user hook = 8 [5 usec]
010   sample     0.000151040  0.005632     USER HOOK 1
                                           The data field for the user hook = 9 [5 usec]
010   sample     0.000156160  0.005120     USER HOOK 1
                                           The data field for the user hook = 10 [5 usec]
```

## Usage Hints

The following sections provide some examples and suggestions for use of the trace facility.

### *Viewing trace Data*

Including several optional columns of data in the trace output can cause the output to exceed 80 columns. It is best to view the report on an output device that supports 132 columns.

You can also use the **-O 2line=on** option to produce a more narrow report.

### *Bracketing Data Collection*

Trace data accumulates rapidly. Bracket the data collection as closely around the area of interest as possible.

One technique for doing this is to issue several commands on the same command line. For example, the command

```
trace -a; cp /etc/trcfmt /tmp/junk; trcstop
```

The following command captures the total copy command run, and stops the trace when the command finishes:

```
trace -ax "cp /etc/trcfmt /tmp/junk"
```

captures the total execution of the copy command.

**Note:** This example is more educational if the source file is not already cached in system memory. The **trcfmt** file can be in memory if you have been modifying it or producing trace reports. In that case, choose as the source file some other file that is 50 to 100 KB and has not been touched.

### *Reading a trace Report*

The trace facility displays system activity. It is a useful learning tool to observe how the system actually performs. The previous output is an interesting example to browse.

To produce a report of the copy, use the following:

```
trcrpt -O "exec=on,pid=on" > cp.rpt
```

In the **cp.rpt** file you can see the following activities:

- The exec and page fault activities of the cp process.
- The fork, exec, and page fault activities of the **cp** process.
- The opening of the **/etc/trcfmt** file for reading and the creation of the **/tmp/junk** file.
- The successive **read** and **write** subroutines to accomplish the copy.
- The **cp** process becoming blocked while waiting for I/O completion, and the wait process being dispatched.
- How logical volume requests are translated to physical volume requests.
- The files are mapped rather than buffered in traditional kernel buffers. The read accesses cause page faults that must be resolved by the virtual memory manager.
- The virtual memory manager senses sequential access and begins to prefetch the file pages.
- The size of the prefetch becomes larger as sequential access continues.
- The writes are delayed until the file is closed (unless you captured execution of the **sync** daemon that periodically forces out modified pages).
- The disk device driver coalesces multiple file requests into one I/O request to the drive when possible.

### *Effective Filtering of the trace Report*

The full detail of the trace data might not be required.

You can choose specific events of interest to be shown. For example, it is sometimes useful to find the number of times a certain event occurred. Answer the question, "How many opens occurred in the copy example?" First, find the event ID for the **open** subroutine:

```
trcrpt -j | pg
```

You can see that event ID 15b is the open event. Now, process the data from the copy example (the data is probably still in the log file) as follows:

```
trcrpt -d 15b -O "exec=on"
```

The report is written to standard output and you can determine the number of opens that occurred. If you want to see only the opens that were performed by the **cp** process, run the report command again using:

```
trcrpt -d 15b -p cp -O "exec=on"
```

This command shows only the opens performed by the **cp** process.

### Trace Event Groups

Combining multiple trace hooks into a trace event group allows all hooks to be turned on or off at once when starting a trace.

Trace event groups should only be manipulated using either the **trcevgrp** command, or SMIT. The **trcevgrp** command allows groups to be created, modified, removed, and listed.

Reserved event groups may not be changed or removed by the **trcevgrp** command. These are generally groups used to perform system support. A reserved event group must be created using the ODM facilities. Such a group will have three attributes as shown below:

```
SWservAt:
       attribute = "(name)_trcgrp"
       default = " "
       value = "(list-of-hooks)"

SWservAt:
       attribute = "(name)_trcgrpdesc"
       default = " "
       value = "description"

SWservAt:
       attribute = "(name)_trcgrptype"
       default = " "
       value = "reserved"
```

The hook IDs must be enclosed in double quotation marks (") and separated by commas.

## Memory Overlay Detection System (MODS)

Some of the most difficult types of problems to debug are what are generally called "memory overlays." Memory overlays include the following:

- Writing to memory that is owned by another program or routine
- Writing past the end (or before the beginning) of declared variables or arrays
- Writing past the end (or before the beginning) of dynamically allocated memory
- Writing to or reading from freed memory
- Freeing memory twice
- Calling memory allocation routines with incorrect parameters or under incorrect conditions.

In the kernel environment (including the kernel, kernel extensions, and device drivers), memory overlay problems have been especially difficult to debug because tools for finding them have not been available. Starting with AIX® 4.2.1, however, the Memory Overlay Detection System (MODS) helps detect memory overlay problems in the kernel, kernel extensions, and device drivers.

**Note:** This feature does not detect problems in application code; it only monitors kernel and kernel extension code.

## bosdebug command

The **bosdebug** command turns the MODS facility on and off. Only the root user can run the **bosdebug** command.

To turn on the base MODS support, type:

```
bosdebug -M
```

For a description of all the available options, type:

```
bosdebug -?
```

After you have run **bosdebug** with the options you want, run the **bosboot -a** command, then shut down and reboot your system (using the **shutdown -r** command). If you need to make any changes to your **bosdebug** settings, you must run **bosboot -a** and **shutdown -r** again.

## When to use the MODS feature

The MODS feature is useful in certain circumstances that are listed in this section.

- When developing your own kernel extensions or device drivers and you want to test them thoroughly.
- When asked to turn on this feature by IBM® technical support service to help in further diagnosing a problem that you are experiencing.

## How MODS works

The primary goal of the MODS feature is to produce a dump file that accurately identifies the problem.

MODS works by turning on additional checking to help detect the conditions listed above. When any of these conditions is detected, your system crashes immediately and produces a dump file that points directly at the offending code. (In previous versions, a system dump might point to unrelated code that happened to be running later when the invalid situation was finally detected.)

If your system crashes while the MODS is turned on, then MODS has most likely done its job.

The **xmalloc** subcommand provides details on exactly what memory address (if any) was involved in the situation, and displays mini-tracebacks for the allocation or free records of this memory.

Similarly, the **netm** command displays allocation and free records for memory allocated using the **net_malloc** kernel service (for example, **mbufs**, **mclusters**, etc.).

You can use these commands, as well as standard crash techniques, to determine exactly what went wrong.

## MODS limitations

There are limitations to the Memory Overlay Detection System. Although it significantly improves your chances, MODS cannot detect all memory overlays. Also, turning MODS on has a variable negative impact (depending on how frequently **xmalloc** or **xmfree** is called) on overall system performance and causes somewhat more memory to be used in the kernel and the network memory heaps.

If your system is running at full processor utilization, or if you are already near the maximums for kernel memory usage, turning on the MODS might cause performance degradation and/or system hangs.

Practical experience with the MODS, however, suggests that the great majority of customers can use it with minimal impact to their systems.

### MODS benefits

You can more easily test and debug your own kernel extensions and device drivers.

In addition, difficult problems that previously required multiple attempts to recreate and debug them generally require many fewer such attempts.

# Loadable Authentication Module Programming Interface

This overview explains the various interfaces for loadable authentication module programming:

## Overview

The loadable authentication module interface provides a means for extending identification and authentication (I&A) for new technologies. The interface implements a set of well-defined functions for performing user and group account access and management.

The degree of integration with the system administrative commands is limited by the amount of functionality provided by the module. When all of the functionality is present, the administrative commands are able to create, delete, modify and view user and group accounts as well as other administrative table data.

The security library and loadable authentication module communicate through the secmethod_table interface. The secmethod_table structure contains a list of subroutine pointers. Each subroutine pointer performs a well-defined operation. These subroutines are used by the security library to perform the operations which would have been performed using the local security database files. There are 2 versions of the secmethod_table structure, version 1 and version 2. Version 1 is now obsolete and applications must be written using the version 2 interface.

**Note:** The version 1 interface can be found in previous versions of this document.

## Load Module Interfaces

Each loadable module defines a number of interface subroutines. The interface subroutines, which must be present, are determined by how the loadable module is to be used by the system.

A loadable module may be used to provide identification (account name and attribute information), authentication (password storage and verification) or both. All modules may have additional support interfaces for initializing and configuring the loadable module, creating new user and group accounts, and serializing access to information. This table describes the purpose of each interface. Interfaces may not be required if the loadable module is not used for the purpose of the interface. For example, a loadable module which only performs authentication functions is not required to have interfaces which are only used for identification operations.

*Table 4. Method Interface Types*

| Method Interface Types | | |
|---|---|---|
| **Name** | **Type** | **Required** |
| `method_attrlist` | Support | No |
| `method_authenticate` | Authentication | No [ 3] |
| `method_chpass` | Authentication | Yes |
| `method_close` | Support | No |
| `method_commit` | Support | No |
| `method_delentry` | Support | No |
| `method_delgroup` | Support | No |
| `method_deluser` | Support | No |

| *Table 4. Method Interface Types (continued)* | | |
|---|---|---|
| **Method Interface Types** | | |
| **Name** | **Type** | **Required** |
| `method_getentry` | Identification [ 1] | No |
| `method_getgracct` | Identification | No |
| `method_getgrgid` | Identification | Yes |
| `method_getgrnam` | Identification | Yes |
| `method_getgrset` | Identification | Yes |
| `method_getgrusers` | Identification | No |
| `method_getpasswd` | Authentication | No |
| `method_getpwnam` | Identification | Yes |
| `method_getpwuid` | Identification | Yes |
| `method_lock` | Support | No |
| `method_newentry` | Support | No |
| `method_newgroup` | Support | No |
| `method_newuser` | Support | No |
| `method_nextentry` | Support | No |
| `method_normalize` | Authentication | No |
| `method_open` | Support | No |
| `method_passwdexpired` | Authentication [ 2] | No |
| `method_passwdrestrictions` | Authentication [ 2] | No |
| `method_putentry` | Identification [ 1] | No |
| `method_putgrent` | Identification | No |
| `method_putgrusers` | Identification | No |
| `method_putpwent` | Identification | No |
| `method_unlock` | Support | No |

**Note:**

1. Any module which provides a *method_attrlist()* interface must also provide this interface.
2. Attributes which are related to password expiration or restrictions should be reported by the *method_attrlist()* interface.
3. If this interface is not provided the *method_getpasswd()* interface must be provided.

Several of the functions make use of a *table* parameter to select between user, group and system identification information. The *table* parameter has one of the following values:

| Identification Table Names | |
|---|---|
| **Value** | **Description** |
| "user" | The table containing user account information, such as user ID, full name, home directory and login shell. |
| "group" | The table containing group account information, such as group ID and group membership list. |
| "system" | The table containing system information, such as user or group account default values. |
| "roles" | The table containing role information such as authorizations, rolelists, and authorization modes. |
| "authorizations" | The table containing authorization information such as explicit role authorizations, authorization children, and authorization description. |
| "privcmds" | The table containing the privileged command information such as command name, security flag, auth privileges, innate privileges, and inherited privileges. |
| "privdevs" | The table containing the privileged devices information such as device name, read privileges, and write privileges. |
| "sysck" | The table containing the sysck information such as class, file type, owner, group, ACLs, and checksum. |

When a *table* parameter is used by an authentication interface, "user" is the only valid value.

## Authentication Interfaces

Authentication interfaces perform password validation and modification.

The authentication interfaces verify that a user is allowed access to the system. The authentication interfaces also maintain the authentication information, typically passwords, which are used to authorize user access.

### The method_authenticate Interface

```
int method_authenticate (void *handlep, char *user, char *response, int
**reenter, char **message);
```

The *method_authenticate* interface verifies that a named user has the correct authentication information, typically a password, for a user account.

The *method_authenticate* interface allocates the message pointer. The message pointer is freed by the calling routine.

The *method_authenticate* interface is called indirectly as a result of calling the authenticate subroutine. The grammar given in the SYSTEM attribute normally specifies the name of the loadable authentication module, but it is not required to do so.

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *user* parameter points to the requested user. The *response* parameter points to the user response to the previous message or password prompt. The *reenter* parameter points to a flag. It is set to a nonzero value when the contents of the *message* parameter must be used as a prompt, and the user's response is used as the *response* parameter when this method is re-invoked. The initial value of the reenter flag is zero. The

*message* parameter points to a character pointer. It is set to a message that is output to the user when an error or warning occurs or an additional prompt is required.

*method_authenticate* returns **AUTH_SUCCESS** with a *reenter* value of zero on success. On failure a value of **AUTH_FAILURE**, **AUTH_UNAVAIL**, **AUTH_NOTFOUND**, **AUTH_PWDMUSTCHANGE**, **AUTH_ACCOUNTLOCKED**, or **AUTH_PWDEXPIRED** is returned.

## The method_chpass Interface

```
int method_chpass (void *handlep, char *user, char *oldpassword, char
*newpassword, char **message);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *user* parameter points to the requested user. The *oldpassword* parameter points to the user's current password. The *newpassword* parameter points to the user's new password. The *message* parameter points to a character pointer. It will be set to a message which is output to the user.

*method_chpass* changes the authentication information for a user account.

*method_chpass* is called indirectly as a result of calling the chpass subroutine. The security library will examine the **registry** attribute for the user and invoke the *method_chpass* interface for the named loadable authentication module.

*method_chpass* returns zero for success or -1 for failure. On failure the *message* parameter should be initialized with a user message.

## The method_getpasswd Interface

```
char *method_getpasswd (void *handlep, char *user);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *user* parameter points to the requested user.

*method_getpasswd* provides the encrypted password string for a user account. The crypt subroutine is used to create this string and encrypt the user-supplied password for comparison.

*method_getpasswd* is called when *method_authenticate* would have been called, but is undefined. The result of this call is compared to the result of a call to the *crypt* subroutine using the response to the password prompt. See the description of the method_authenticate interface for a description of the *response* parameter.

*method_getpasswd* returns a pointer to an encrypted password on success. On failure a **NULL** pointer is returned and the global variable **errno** is set to indicate the error. A value of **ENOSYS** is used when the module cannot return an encrypted password. A value of **EPERM** is used when the caller does not have the required permissions to retrieve the encrypted password. A value of **ENOENT** is used when the requested user does not exist.

## The method_normalize Interface

```
int method_normalize (void *handlep, char *longname, char *shortname);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *longname* parameter points to a fully-qualified user name for modules which include domain or registry information in a user name. The *shortname* parameter points to the shortened name of the user, without the domain or registry information.

*method_normalize* determines the shortened user name which corresponds to a fully-qualified user name. The shortened user name is used for user account queries by the security library. The fully-qualified user name is only used to perform initial authentication.

If the fully-qualified user name is successfully converted to a shortened user name, a non-zero value is returned. If an error occurs a zero value is returned.

### The method_passwdexpired Interface

```
int method_passwdexpired (void *handlep, char *user, char **message);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *user* parameter points to the requested user. The *message* parameter points to a character pointer. It will be set to a message which is output to the user.

*method_passwdexpired* determines if the authentication information for a user account is expired. This method distinguishes between conditions which allow the user to change their information and those which require administrator intervention. A message is returned which provides more information to the user.

*method_passwdexpired* is called as a result of calling the passwdexpired subroutine.

*method_passwdexpired* returns 0 when the password has not expired, 1 when the password is expired and the user is permitted to change their password and 2 when the password has expired and the user is not permitted to change their password. A value of -1 is returned when an error has occurred, such as the user does not exist.

### The method_passwdrestrictions Interface

```
int method_passwdrestrictions (void *handlep, char *user, char *newpassword,
char *oldpassword, char **message);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *user* parameter points to the requested user. The *newpassword* parameter points to the user's new password. The *oldpassword* parameter points to the user's current password. The *message* parameter points to a character pointer. It will be set to a message which is output to the user.

*method_passwdrestrictions* determines if new password meets the system requirements. This method distinguishes between conditions which allow the user to change their password by selecting a different password and those which prevent the user from changing their password at the present time. A message is returned which provides more information to the user.

*method_passwdrestrictions* is called as a result of calling the security library subroutine *passwdrestrictions*.

*method_passwdrestrictions* returns a value of 0 when *newpassword* meets all of the requirements, 1 when the password does not meet one or more requirements and 2 when the password may not be changed. A value of -1 is returned when an error has occurred, such as the user does not exist.

## Identification Interfaces

Identification interfaces perform user and group identity functions. The identification interfaces store and retrieve user and group identifiers and account information.

The identification interfaces divide information into three different categories: user, group, and system. User information consists of the user name, user and primary group identifiers, home directory, login shell and other attributes specific to each user account. Group information consists of the group identifier, group member list, and other attributes specific to each group account. System information consists of default values for user and group accounts, and other attributes about the security state of the current system.

### The method_getentry Interface

```
int method_getentry (void *handlep, char *key, char *table, dbattr_t results[],
int size);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *key* parameter refers to an entry in the named table. The *table* parameter refers to one of the tables. The *results* parameter refers to an array of data structures, which contains the names of the attributes and their types to be returned by this method, as well as either the value of the corresponding attribute being returned or a flag indicating a cause of failure. The *size* parameter is the number of array elements.

*method_getentry* retrieves user, group, and system attributes. One or more attributes may be retrieved for each call. Success or failure is reported for each attribute.

*method_getentry* is called as a result of calling the getuserattrs andgetgroupattrs subroutines.

*method_getentry* returns a value of 0 if the *key* entry was found in the named *table*. When the entry does not exist in the table, the global variable **errno** must be set to **ENOENT**. If an error in the value of *table* or *size* is detected, the **errno** variable must be set to **EINVAL**. Individual attribute values have additional information about the success or failure for each attribute. On failure a value of -1 is returned.

## The method_getgracct Interface

```
struct group *method_getgracct (void *handlep, void *id, int type);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *id* parameter refers to a group name or GID value, depending upon the value of the *type* parameter. The *type* parameters indicates whether the *id* parameter is to be interpreted as a `(char *)` which references the group name, or `(gid_t)` for the group.

*method_getgracct* retrieves basic group account information. The *id* parameter may be a group name or identifier, as indicated by the *type* parameter. The basic group information is the group name and identifier. The group member list is not returned by this interface.

*method_getgracct* may be called as a result of calling the IDtogroup subroutine.

*method_getgracct* returns a pointer to the group's group file entry on success. The group file entry may not include the list of members. On failure a **NULL** pointer is returned.

## The method_getgrgid Interface

```
struct group *method_getgrgid (void *handlep, gid_t gid);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *gid* parameter is the group identifier for the requested group.

*method_getgrgid* retrieves group account information given the group identifier. The group account information consists of the group name, identifier, and complete member list.

*method_getgrgid* is called as a result of calling the getgrgid subroutine.

*method_getgrgid* returns a pointer to the group's group file structure on success. On failure a **NULL** pointer is returned.

## The method_getgrnam Interface

```
struct group *method_getgrnam (void *handlep, const char *group);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *group* parameter points to the requested group.

*method_getgrnam* retrieves group account information given the group name. The group account information consists of the group name, identifier, and complete member list.

*method_getgrnam* is called as a result of calling the getgrnam subroutine. This interface may also be called if *method_getentry* is not defined.

*method_getgrnam* returns a pointer to the group's group file structure on success. On failure a **NULL** pointer is returned.

## The method_getgrset Interface

```
char *method_getgrset (void *handlep, char *user);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *user* parameter points to the requested user.

*method_getgrset* retrieves supplemental group information given a user name. The supplemental group information consists of a comma separated list of group identifiers. The named user is a member of each listed group.

*method_getgrset* is called as a result of calling the getgrset subroutine.

*method_getgrset* returns a pointer to the user's concurrent group set on success. On failure a **NULL** pointer is returned.

## The method_getgrusers Interface

```
int method_getgrusers (void *handlep, char *group, void *result, int type, int
*size);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *group* parameter points to the requested group. The *result* parameter points to a storage area which will be filled with the group members. The *type* parameters indicates whether the *result* parameter is to be interpreted as a `(char **)` which references a user name array, or `(uid_t)` array. The *size* parameter is a pointer to the number of users in the named group. On input it is the size of the *result* field.

*method_getgrusers* retrieves group membership information given a group name. The return value may be an array of user names or identifiers.

*method_getgrusers* may be called by the security library to obtain the group membership information for a group.

*method_getgrusers* returns 0 on success. On failure a value of -1 is returned and the global variable **errno** is set. The value **ENOENT** must be used when the requested group does not exist. The value **ENOSPC** must be used when the list of group members does not fit in the provided array. When **ENOSPC** is returned the *size* parameter is modified to give the size of the required *result* array.

## The method_getpwnam Interface

```
struct passwd *method_getpwnam (void *handlep, const char *user);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *user* parameter points to the requested user.

*method_getpwnam* retrieves user account information given the user name. The user account information consists of the user name, identifier, primary group identifier, full name, login directory and login shell.

*method_getpwnam* is called as a result of calling the getpwnam subroutine. This interface may also be called if *method_getentry* is not defined.

*method_getpwnam* returns a pointer to the user's password structure on success. On failure a **NULL** pointer is returned.

## The method_getpwuid Interface

```
struct passwd *method_getpwuid (void *handlep, uid_t uid);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *uid* parameter points to the user ID of the requested user.

*method_getpwuid* retrieves user account information given the user identifier. The user account information consists of the user name, identifier, primary group identifier, full name, login directory and login shell.

*method_getpwuid* is called as a result of calling the getpwuid subroutine.

*method_getpwuid* returns a pointer to the user's password structure on success. On failure a **NULL** pointer is returned.

## The method_putentry Interface

```
int method_putentry (void *handlep, char *key, char *table, dbattr_t results[],
int size);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *key* parameter refers to an entry in the named table. The *table* parameter refers to one of the tables. The *attributes* parameter refers to an array of pointers to attribute names. The *values* parameter refers to an array of value structures which correspond to the attributes. Each value structure contains a flag indicating if the attribute was output. The *size* parameter is the number of array elements.

*method_putentry* stores user, group and system attributes. One or more attributes may be output for each call. Success or failure is reported for each attribute. Values will be saved until method_commit is invoked.

*method_putentry* is called as a result of calling the putuserattrs, putgroupattrs, and putconfattrs subroutines.

*method_putentry* returns 0 when the attributes have been updated. On failure a value of -1 is returned and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when updating information is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to create the group. A value of **ENOENT** is used when the entry does not exist. A value of **EROFS** is used when the module was not opened for updates.

## The method_putgrent Interface

```
int method_putgrent (void *handlep, struct group *entry);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *entry* parameter points to the structure to be output. The account name is contained in the structure.

*method_putgrent* stores group account information given a group entry. The group account information consists of the group name, identifier and complete member list. Values will be saved until method_commit is invoked.

*method_putgrent* returns 0 when the group has been successfully updated. On failure a value of -1 is returned and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when updating groups is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to update the group. A value of **ENOENT** is used when the group does not exist. A value of **EROFS** is used when the module was not opened for updates.

## The method_putgrusers Interface

```
int method_putgrusers (void *handlep, char *group, char *users);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *group* parameter points to the requested group. The *users* parameter points to a **NULL** character separated, double **NULL** character terminated, list of group members.

*method_putgrusers* stores group membership information given a group name. Values will be saved until method_commit is invoked.

*method_putgrusers* returns 0 when the group has been successfully updated. On failure a value of -1 is returned and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when updating groups is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to update the group. A value of **ENOENT** is used when the group does not exist. A value of **EROFS** is used when the module was not opened for updates.

## The method_putpwent Interface

```
int method_putpwent (void *handlep, struct passwd *entry);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *entry* parameter points to the structure to be output. The account name is contained in the structure.

*method_putpwent* stores user account information given a user entry. The user account information consists of the user name, identifier, primary group identifier, full name, login directory and login shell. Values will be saved until method_commit is invoked.

*method_putpwent* returns 0 when the user has been successfully updated. On failure a value of -1 is returned and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when updating users is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to update the user. A value of **ENOENT** is used when the user does not exist. A value of **EROFS** is used when the module was not opened for updates.

## Support Interfaces

Support interfaces perform functions such as initiating and terminating access to the module, creating and deleting accounts, and serializing access to information.

### The method_attrlist Interface

```
attrtab **method_attrlist (void *handlep);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. *method_attrlist* provides a means of defining attributes for a loadable module. Authentication-only modules may use this interface to override attributes which would normally come from the identification module half of a compound load module.

*method_attrlist* is called when a loadable module is first initialized. The return value will be saved for use by later calls to various identification and authentication functions.

### The method_close Interface

```
void method_close (void *handlep);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. *method_close* indicates that access to the loadable module has ended and all system resources may be freed. The loadable module must not assume this interface will be invoked as a process may terminate without calling this interface.

*method_close* is called when the session count maintained by enduserdb reaches zero.

There are no defined error return values. It is expected that the *method_close* interface handle common programming errors, such as being invoked with an invalid token, or repeatedly being invoked with the same token.

### The method_commit Interface

```
int method_commit (void *handlep, char *key, char *table);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *key* parameter refers to an entry in the named table. If it is **NULL** it refers to all entries in the table. The *table* parameter refers to one of the tables.

*method_commit* indicates that the specified pending modifications are to be made permanent. An entire table or a single entry within a table may be specified. method_lock will be called prior to calling *method_commit*. method_unlock will be called after *method_commit* returns.

*method_commit* is called when putgroupattr or putuserattr are invoked with a *Type* parameter of **SEC_COMMIT**. The value of the *Group* or *User* parameter will be passed directly to *method_commit*.

*method_commit* returns a value of 0 for success. A value of -1 is returned to indicate an error and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when the load module does not support modification requests for any users. A value of **EROFS** is used when the module is not currently opened for updates. A value of **EINVAL** is used when the *table* parameter refers to an invalid table. A value of **EIO** is used when a potentially temporary input-output error occurs.

## The method_delgroup Interface

```
int method_delgroup (void *handlep, char *group);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *group* parameter points to the requested group.

*method_delgroup* removes a group account and all associated information. A call to *method_commit* is not required. The group will be removed immediately.

*method_delgroup* is called when putgroupattr is invoked with a *Type* parameter of **SEC_DELETE**. The value of the *Group* and *Attribute* parameters will be passed directly to *method_delgroup*.

*method_delgroup* returns 0 when the group has been successfully removed. On failure a value of -1 is returned and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when deleting groups is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to delete the group. A value of **ENOENT** is used when the group does not exist. A value of **EROFS** is used when the module was not opened for updates. A value of **EBUSY** is used when the group has defined members.

## The method_deluser Interface

```
int method_deluser (void *handlep, char *user, char *attrname);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *user* parameter points to the requested user. The *attrname* parameter indicates the attribute name of the user to be deleted.

*method_deluser* removes a single user's attribute if *attrname* is supplied or it removes a user's account and all associated information if just the user name is supplied. A call to *method_commit* is not required. The user will be removed immediately.

*method_deluser* is called when putuserattr is invoked with a *Type* parameter of **SEC_DELETE**. The value of the *User* and *Attribute* parameters will be passed directly to *method_deluser*.

*method_deluser* returns 0 when the user has been successfully removed. On failure a value of -1 is returned and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when deleting users is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to delete the user. A value of **ENOENT** is used when the user does not exist. A value of **EROFS** is used when the module was not opened for updates.

## The method_lock Interface

```
void *method_lock (void *handlep, char *key, char *table, int wait);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *key* parameter refers to an entry in the named table. If it is **NULL** it refers to all entries in the table. The *table* parameter refers to one of the tables. The *wait* parameter is the number of second to wait for the lock to be acquired. If the *wait* parameter is zero the call returns without waiting if the entry cannot be locked immediately.

*method_lock* informs the loadable modules that access to the underlying mechanisms should be serialized for a specific table or table entry.

*method_lock* is called by the security library when serialization is required. The return value will be saved and used by a later call to method_unlock when serialization is no longer required.

## The method_newentry Interface

```
int method_newentry (void *handlep, char *table_name, char *key, char
*attrname);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *key* parameter refers to an entry in the named table. If it is **NULL** it refers to all entries in the table. The

*table_name* parameter refers to one of the tables. The *attrname* parameter indicates the attribute name which the routine uses to determine what files need to be accessed.

*method_newentry* creates an entry in a table other than the **user** or **group** table. The entry information will not be made permanent until method_commit is invoked.

*method_newentry* returns 0 when the entry has been successfully created. On failure a value of -1 is returned and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when creating an entry is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to create an entry. A value of **EEXIST** is used when the entry already exists. A value of **EROFS** is used when the module was not opened for updates.

## The method_newgroup Interface

```
int method_newgroup (void *handlep, char *group);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *group* parameter points to the requested group.

*method_newgroup* creates a group account. The basic group account information must be provided with calls to method_putgrent or method_putentry. The group account information will not be made permanent until method_commit is invoked.

*method_newgroup* is called when putgroupattr is invoked with a *Type* parameter of **SEC_NEW**. The value of the *Group* parameter will be passed directly to *method_newgroup*.

*method_newgroup* returns 0 when the group has been successfully created. On failure a value of -1 is returned and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when creating group is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to create the group. A value of **EEXIST** is used when the group already exists. A value of **EROFS** is used when the module was not opened for updates. A value of **EINVAL** is used when the group has an invalid format, length or composition.

## The method_newuser Interface

```
int method_newuser (void *handlep, char *user);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *user* parameter points to the requested user.

*method_newuser* creates a user account. The basic user account information must be provided with calls to method_putpwent or method_putentry. The user account information will not be made permanent until method_commit is invoked.

*method_newuser* is called when putuserattr is invoked with a *Type* parameter of **SEC_NEW**. The value of the *User* parameter will be passed directly to *method_newuser*.

*method_newuser* returns 0 when the user has been successfully created. On failure a value of -1 is returned and the global variable **errno** is set to indicate the cause. A value of **ENOSYS** is used when creating users is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to create the user. A value of **EEXIST** is used when the user already exists. A value of **EROFS** is used when the module was not opened for updates. A value of **EINVAL** is used when the user has an invalid format, length, or composition.

## The method_nextentry Interface

```
int method_newentry (void *handlep, char *table_name, char *attrname, char
*key);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *key* parameter refers to an entry in the named table. If it is NULL it refers to all entries in the table. The table_name parameter refers to one of the tables. The *attrname* parameter indicates the attribute name which the routine uses to determine what files need to be accessed.

*method_nextentry* locates the name of the next entry in a linear search through a file given the previous search entry name and the name of an attribute which selects a specific file.

method_nextentry returns 0 when the entry has been successfully created. On failure a value of -1 is returned and the global variable errno is set to indicate the cause. A value of **EINVAL** is used when an invalid table name has been specified, or if no attrname has been specified and the table is not one of these tables. A value of **ENOSYS** is used when creating an entry is not supported by the module. A value of **EPERM** is used when the invoker does not have permission to create an entry. A value of **EEXIST** is used when the entry already exists. A value of **EROFS** is used when the module was not opened for updates.

### The method_open Interface

```
void *method_open (char *name, char *domain, int mode, char *options);
```

The *name* parameter is a pointer to the stanza name in the configuration file. The *domain* parameter is the value of the **domain=** attribute in the configuration file. The *mode* parameter is either **O_RDONLY** or **O_RDWR**. The *options* parameter is a pointer to the **options=** attribute in the configuration file.

*method_open* prepares a loadable module for use. The domain and options attributes are passed to *method_open*.

*method_open* is called by the security library when the loadable module is first initialized and when setuserdb is first called after method_close has been called due to an earlier call to enduserdb. The return value will be saved for a future call to method_close.

### The method_unlock Interface

```
void method_unlock (void *handlep, void *token);
```

The *handlep* parameter is a handle to the load module opened with the method_open interface. The *token* parameter is the value of the corresponding *method_lock* call.

*method_unlock* informs the loadable modules that an earlier need for access serialization has ended.

*method_unlock* is called by the security library when serialization is no longer required. The return value from the earlier call to method_lock be used.

## Configuration Files

A stanza exists for each loadable module which is to be used by the system. Each stanza contains a number of attributes used to load and initialize the module.

The security library uses the /etc/methods.cfg file to control which modules are used by the system. The loadable module may use this information to configure its operation when the *method_open()* interface is invoked immediately after the module is loaded.

### The options Attribute

The entire value of the *options* attribute is passed to the *method_open()* subroutine when the module is first initialized. Five pre-defined flags control how the library uses the loadable module.

The options attribute will be passed to the loadable module when it is initialized. This string is a comma-separated list of *Flag* and *Flag=Value* entries.

| Item | Description |
|---|---|
| **auth=***module* | *Module* will be used to perform authentication functions for the current loadable authentication module. Subroutine entry points dealing with authentication-related operations will use method table pointers from the named module instead of the module named in the *program=* or *program_64=* attribute. |

| Item | Description |
|---|---|
| **authonly** | The loadable authentication module only performs authentication operations. Subroutine entry points which are not required for authentication operations, or general support of the loadable module, will be ignored. |
| **db=***module* | *Module* will be used to perform identification functions for the current loadable authentication module. Subroutine entry points dealing with identification related operations will use method table pointers from the name module instead of the module named in the *program=* or *program_64=* attribute. |
| **dbonly** | The loadable authentication module only provides user and group identification information. Subroutine entry points which are not required for identification operations, or general support of the loadable module, will be ignored. |
| **noprompt** | The initial password prompt for authentication operations is suppressed. Password prompts are normally performed prior to a call to *method_authenticate()*. *method_authenticate()* must be prepared to receive a **NULL** pointer for the *response* parameter and set the *reenter* parameter to **TRUE** to indicate that the user must be prompted with the contents of the *message* parameter prior to *method_authenticate()* being re-invoked. See the description of method_authenticate for more information on these parameters. |

## Compound Load Modules

The security library is responsible for constructing a new method table to perform the compound function.

Compound load modules are created with the auth= and db= attributes.

Interfaces are divided into three categories: identification, authentication and support. Identification interfaces are used when a compound module is performing an identification operation, such as the getpwnam subroutine. Authentication interfaces are used when a compound module is performing an authentication operation, such as the authenticate subroutine. Support subroutines are used when initializing the loadable module, creating or deleting entries, and performing other non-data operations. The table Method Interface Types describes the purpose of each interface. The table below describes which support interfaces are called in a compound module and their order of invocation.

| Support Interface Invocation | |
|---|---|
| **Name** | **Invocation Order** |
| `method_attrlist` | Identification, Authentication |
| `method_close` | Identification, Authentication |
| `method_commit` | Identification, Authentication |
| `method_deluser` | Authentication, Identification |
| `method_lock` | Identification, Authentication |
| `method_newuser` | Identification, Authentication |
| `method_open` | Identification, Authentication |
| `method_unlock` | Authentication, Identification |

**Related information**

Identification and Authentication Subroutines

/etc/methods.cfg File

# Kernel Storage-Protection Keys

Kernel storage-protection keys provide a storage-protection mechanism for the kernel and kernel extensions.

Storage protection keys have the following elements:

- Each virtual memory page is assigned a small integer protection key, which represents a required access authority to make the page readable or writable in the current context.
- A register, the authority mask register, is used to encode the access authority of the current context.

The hardware can provide up to 32 protection keys. The authority mask register contains a pair of bits representing the write and read access authorities for each of the defined protection keys, making it possible for the kernel to control access to as many as 32 classes of memory, where each class consists of virtual memory pages sharing the same specific protection key.

In all cases, a running thread can modify its authority mask register, and hence its authority to access specific memory classes. In AIX® environment, the storage-protection mechanism can catch programming errors caused by inadvertent storage overlays. When you write a program using storage protection keys, the program can voluntarily subdivide its memory for protection purposes, decreasing the likelihood that an accidental reference to memory goes undetected. Such undetected errors can be difficult to debug; however, by coding to detect them as they occur, a program can improve its reliability, availability, and serviceability characteristics.

When you use storage protection keys in the kernel, the following programming errors can be detected and easily repaired:

- Incorrect reference to application memory by the kernel
- Incorrect reference to kernel private memory by key-safe kernel extensions
- Incorrect reference to kernel extension private memory by the kernel
- Incorrect reference to memory controlled by one kernel subsystem by another

Kernel extensions can be classified into the following categories from the storage protection aspect:

**Key-unsafe kernel extensions**
Key-unsafe kernel extensions are traditional extensions that were written without regard to storage key protection. When loaded in a key-safe environment, such extensions are automatically run with wide access authority to provide binary compatibility. With the ability to access essentially all kernel memory without restrictions, key-unsafe kernel extensions can continue to function, even though they might violate the protection domain of the kernel.

In some other cases key-unsafe kernel extensions can cause system crashes therefore it is highly recommended that kernel extensions should be made key-safe before running. See “Making a Kernel Extension Key Safe” on page 424.

**Key-safe kernel extensions**
Key-safe kernel extensions do not directly refer to either the internal data structures of the kernel or user space addresses. See “Making a Kernel Extension Key Safe” on page 424.

**Key-protected kernel extensions**
Key-protected kernel extensions can coexist with the key-protected kernel environment, and also become key-protected by identifying and protecting their own private data. To place kernel extension data in key-protected memory, see “Designing the Key Protection in a Key-protected Kernel Extension” on page 424.

**Related information**
kkey_assign_private
kkeyset_create
kkeyset_add_key
kkeyset_remove_key
kkeyset_add_set

kkeyset_remove_set
kkeyset_delete
kkeyset_to_hkeyset
hkeyset_add, hkeyset_replace, hkeyset_restore, or hkeyset_get Kernel Service

## Kernel Keys and Kernel Key Sets

A kernel key is a virtual key that is assigned by the kernel. Kernel programs can use more virtual keys than exist in the hardware, because many kernel keys can share a single hardware key.

The kernel data is classified into keys according to function. You can use kernel keys to define a large number of virtual storage protection keys. Most kernel keys are used only within the kernel. The **sys/skeys.h** file contains a list of all keys. The following kernel keys can be useful in your kernel extensions:

**KKEY_PUBLIC**
Used to access stack, bss and data.

**KKEY_BLOCK_DEV**
Required for block device drivers. The **buf** structures must be either public or in the key.

**KKEY_COMMO**
Required for communication drivers.

**KKEY_NETM**
Required for drivers to reference memory that is allocated by the **net_malloc** kernel service.

**KKEY_USB**
Required for USB device drivers.

**KKEY_GRAPHICS**
Required for graphics device drivers.

**KKEY_DMA**
Required for direct memory access (DMA) information.

**KKEY_TRB**
Required for timer services (the **trb** structure).

**KKEY_IOMAP**
Used to access I/O mapped segments.

**KKEY_FILE_SYSTEM**
Used to access vnodes and gnodes (**vnop** callers).

You can use a set of keys for a typical kernel extension using one of the predefined kernel key sets. The **sys/skeys.h** file contains a list of all key sets. The following predefined kernel key sets are useful with read and write accesses to the member kernel keys. Similar sets whose names are appended with **_READ** grant only read access.

**KKEYSET_GLOBAL**
All known kernel keys

**KKEYSET_LEGACY**
The large set of kernel keys that is granted to kernel extensions that are not key-safe

**KKEYSET_KERNEXT**
The kernel keys common to most kernel extensions

**KKEYSET_COMMO**
The common keys plus those that are used by communication drivers

**KKEYSET_BLOCK**
The common keys plus those that are used by block I/O drivers

**KKEYSET_GRAPHICS**
The common keys plus those that are used by graphics drivers

**KKEYSET_USB**
The common keys plus those that are used by USB drivers

**KKEYSET_USERDATA**
All known user-space kernel keys

## Protection Gates

To make a kernel extension key-safe or key-protected, you must add protection gates, typically at all entry and exit points of your module. With proper protection gates, your module has access to the data that it requires, and does not have access to the data that it does not require.

You can specify the following gates at an entry point:

**An add gate**
With an add gate, you can augment the callers key set with your own. You can specify an add gate for a service where your caller passes pointers to data. The data of the caller might be protected with a key you are unaware of, therefore you must retain these keys to refer to the data of the caller. You can add additional keys so that you can also refer to any private data that you need.

**A replace gate**
With a replace gate, you can switch to your own key set. You can specify a replace gate at an entry point for a callback that you have registered with, for example, the device switch table. You can also specify the replace gate to relinquish the keys of the caller, when the kernel is your caller. (Do not retain access to internal data of the kernel.) You can use the predefined key sets as described in “Kernel Keys and Kernel Key Sets” on page 422 to form the basis of a typical replace gate.

In both cases, the protection gate service returns the original authority mask register value to you, so that you can restore keys at your exit points.

You can decide where to place these gates in your program flow. You can identify and place gates at all entry points that are externally visible. One common exception is to defer the gate when you take advantage of keys of the caller to pick up potentially private data being passed in. Then switch to your own key set with a replace gate. This technique provides better storage protection than adding an add gate right at the entry point.

**Important:** Remember to restore the necessary keys at any exit point, where data might be stored back through a parameter pointer.

You can use the following kernel services to implement your protection gates:

**hkeyset_add**
Adds access rights to the currently addressable hardware key set, and returns the old hardware key set.

**hkeyset_replace**
Loads a new set of access rights, and returns the old hardware key set.

**hkeyset_restore**
Restores a saved set of access rights.

**hkeyset_get**
Returns the current access rights.

To identify the entry points of your kernel extension, look for the following typical ones:

- Device-switch table callbacks for open, close, read, write, ioctl, strategy, select, print, dump, mpx, and revoke entry points

  **Tip:** Typically, the config entry point does not need a protection gate, but includes the initialization necessary for protection gates, heaps and so on, for subsequent uses by other entry points.
- The timer event handler
- The watchdog handler
- The enhanced I/O error handling handlers
- The interrupt handler (INTCLASSx, iodone, offlevel)
- The environmental and power warning handler
- Exported system calls

- The dynamic reconfiguration and high availability handlers
- The shutdown notification handler
- The RAS callback
- Streams callback functions
- Process state change notification handlers
- Function pointers that are passed outside of your module

## Making a Kernel Extension Key Safe

Key-safe kernel extensions do not directly refer to either the internal data structures of the kernel or user space addresses.

To make a kernel extension key-safe, follow these steps:

1. Decide which kernel key set, if any, can be the basis for the key set of your module.
2. Optionally, remove any unnecessary keys from your copy of the kernel key set.
3. Convert the kernel key set to a hardware key set.
4. Place add and replace protection gates at or near all entry points (except initialization). See "Protection Gates" on page 423.
5. Place restore gates at or near exit points.
6. Link your extension with the new **-b ras** flag to identify the extension to the system as reliability, availability, and serviceability aware.

   **Restriction:** You must specify **-q noinlglue** to ensure that the compiler does not generate inline pointer glue.

## Designing the Key Protection in a Key-protected Kernel Extension

You must design the key protection so that any incorrect references to memmory can easily be detected and repaired.

To design key protection into a kernel extension, perform the following steps:

1. Define memory classes. A memory class is a set of virtual pages that are associated with the same storage protection key. Ideally, each memory class is represented by a unique kernel key, although there are limits on both the actual number of kernel keys that can be used, and the actual number of hardware keys that are available for the kernel and kernel extensions.
2. Allocate a kernel key for each memory class. You can use the **kkey_assign_private** kernel service to allocate unique kernel keys.
3. Assign kernel keys to the (pages of) memory to be included in each memory class. You can use the **heap_create** kernel service to create a memory heap whose pages are protected with a specified kernel key. Your new heap can then be used by the **xmalloc** and **xmfree** kernel services as usual.

   When your program ends, use the **heap_destroy** kernel service to clean up any heaps you have created.
4. Determine which kernel keys are required by the kernel extension to satisfy the requirements of other kernel extension data that is referred to. For the best result, add the access rights from one of these predefined key sets to your own, rather than build up your key set from scratch. See "Kernel Keys and Kernel Key Sets" on page 422.
5. Create the kernel key sets defining the access authorities that are needed by your extension to refer to its private memory classes. You can use the following kernel services:

   **kkeyset_create**
   Allocates an empty key set.

   **kkeyset_add_key**
   Adds access rights for a kernel key to your key set.

**kkeyset_remove_key**
Removes access rights to a given kernel key from your key set.

**kkeyset_add_set**
Adds the access rights that are defined in one key set to another.

**kkeyset_remove_set**
Removes the access rights that are defined in one key set from another.

**kkeyset_delete**
Frees a key set after it has been used to create a hardware key set by "6" on page 425.

6. Create hardware key sets that you need to control access to the memory classes during execution. These hardware key sets are needed to implement your protection gates. The **kkeyset_to_hkeyset** service performs this translation.

   **Attention:** The **HKEYSET_GLOBAL** key set is defined to access any memory no matter which key protects it. Therefore, this hardware key set should be used with caution.

7. Implement protection gates as needed to grant access to the memory classes, and to restore the prior authority mask register state. Typically, you can perform these steps at the entry and exit points of the major functions of the kernel extension. You can implement a replace gate at an entry point in your driver to grant access appropriately. Without one, the driver has the inappropriate access rights of its caller, typically the kernel. You can implement a restore gate at exit points to restore the access rights back to those of the calling program. See "Protection Gates" on page 423 for more information about protection gates.
8. Link the kernel extension using the new **-b ras** flag, so that the kernel can recognize it as a key-safe kernel extension.

If the extension must refer to user space memory, use the **copyin**, **copyout**, or one of the cross-memory kernel services. These services acquire the appropriate additional hardware keys necessary to access and protect user space. (The user-space application might use storage protect keys itself.) It is possible, but not suggested, to add user keys to the current authority mask register with the **hkeyset_update_userkeys** service, and to restore the previous state with the **hkeyset_restore_userkeys** service.

Any kernel keys can be used, with appropriate care, by any kernel extension. You can also remove keys that you do not need. For example, you can remove the KKEY_TRB key if your extension does not use services such as the **tstart** kernel service, which uses the **trb** structure. Though not required, the action can increase the benefit of using storage protect keys by minimizing the access rights of your kernel extension. Start with a predefined key set, and remove specific keys; rather than start with an empty key set, and figure out which ones you need to add.

In general, you do not need to test your code for the presence, absence, or number of available hardware keys. All key-protection kernel services work efficiently in all cases.

## Examples

The following examples provide a guide for designing key protection into a kernel extension.

1. The following example creates a pinned, shared heap that is assigned to a specified kernel key.

```
#include <sys/malloc.h>
#include <sys/skeys.h>
kkey_t my_kkey;
heapattr_t heapattr;
heapaddr_t my_heap = HPA_INVALID_HEAP

rc = kkey_assign_private("my string", 0, 0, &my_kkey);
bzero(&heapattr, sizeof(heapattr));
heapattr.hpa_eyec = EYEC_HEAPATTR;
heapattr.hpa_version = HPA_VERSION;
heapattr.hpa_flags = HPA_PINNED | HPA_SHARED;
heapattr.hpa_debug_level = HPA_DEFAULT_DEBUG;
heapattr.hpa_kkey = my_kkey;
rc = heap_create(&heapattr, &my_heap);
```

2. The following examples create a hardware key set.
   - Initialize a hardware key set (errors ignored for brevity)

```
#include <sys/skeys.h>
hkeyset_t my_hkeyset;    /* globally visible hardware keyset */
kkeyset_t my_kkeyset;    /* temporary kernel keyset */

my_kkeyset = KKEYSET_INVALID;
rc = kkeyset_create(&my_kkey);
rc = kkeyset_add_set(my_kkeyset, KKEYSET_BLOCK);
rc = kkeyset_add_key(my_kkeyset, my_kkey, KA_RW);
rc = kkeyset_to_hkeyset(my_kkeyset, &my_hkeyset);
kkeyset_delete(my_kkeyset);
```

   - Implement a protection gate

```
hkeyset_t old_hkeyset;
old_hkeyset = hkeyset_replace(my_hkeyset);
… …
hkeyset_restore(old_hkeyset);
```

# Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing*
*IBM Corporation*
*North Castle Drive, MD-NC119*
*Armonk, NY 10504-1785*
*US*

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing*
*Legal and Intellectual Property Law*
*IBM Japan Ltd.*
*19-21, Nihonbashi-Hakozakicho, Chuo-ku*
*Tokyo 103-8510, Japan*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*IBM Director of Licensing*
*IBM Corporation*
*North Castle Drive, MD-NC119*
*Armonk, NY 10504-1785*
*US*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

## Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

## Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

UNIX is a registered trademark of The Open Group in the United States and other countries.

# Index

## D

## E



## F



## G

## H



## I



## K

## L



## M

## N



## O



## P

## R



## S

## T



## U

## V