AIX Version 3 for
RISC System/6000™

Kernel Extensions and Device Support Programming Concepts

SC23-2

RISC System/6000™

Kernel Extensions and Device Support Programming Concepts

First Edition (March 1990)

This edition of the *AIX Kernel Extensions and Device Support Programming Concepts for IBM RISC System/6000* applies to Version 3 of the IBM AIX Base Operating System Licensed Program and to all subsequent releases of this product until otherwise indicated in new releases or technical newsletters.

Trademarks and Acknowledgements

About This Book

This book, *AIX Kernel Extensions and Device Support Programming Concepts for IBM RISC System/6000,* provides a conceptual introduction to the kernel programming environment and writing kernel extensions. Possible types of kernel extensions include device drivers, system calls, kernel services, or virtual file systems. In addition, conceptual information is provided on existing kernel subsystems.

More detailed information on existing kernel services and interface requirements for kernel extensions can be found in *AIX Calls and Subroutines Reference for IBM RISC System/6000*, Volume 5.

Who Should Use This Book

This book is intended for systems programmers wishing to extend the AIX kernel. Readers should be familiar with operating system concepts and kernel programming.

How to Use This Book

Overview of Contents

The *Kernel Extensions and Device Support Programming Concepts* contains two parts. Part 1 contains information needed to write kernel extensions. This includes:

- An overview of the kernel programming environment
- Conceptual introductions to device drivers, system calls, and virtual file systems.

Part 2 gives an overview of AIX subsystems and describes the organization of each. Conceptual information on the following AIX subsystems is provided:

- The communications I/O subsystem. This chapter describes features common to all communications device drivers.
- The configuration subsystem. This chapter includes an overview of the configuration process, the routines and databases involved, and the requirements for configuring new devices.
- The high function terminal (HFT) subsystem. This chapter describes the component structure of the high function terminal and virtual terminal concepts.
- The printer addition management subsystem. This chapter briefly describes the steps involved in adding a new type of printer to the system.
- The SCSI subsystem. This chapter briefly discusses SCSI subsystem architecture and general comments about writing SCSI device drivers.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
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.

Related Publications

The following books contain information about or related to the kernel programming environment and writing kernel extensions:

- *AIX Calls and Subroutines Reference for IBM RISC System/6000*, Order Number SC23–2198.
- *General Programming Concepts*, Order Number SC23–2205–0.
- *Communications Programming Concepts*, Order Number SC23–2206–0.
- *Graphics Programming Concepts*, Order Number SC23–2208–0.

Ordering Additional Copies of This Book

To order additional copies of this book, use Order Number SC23–2207.

Table of Contents

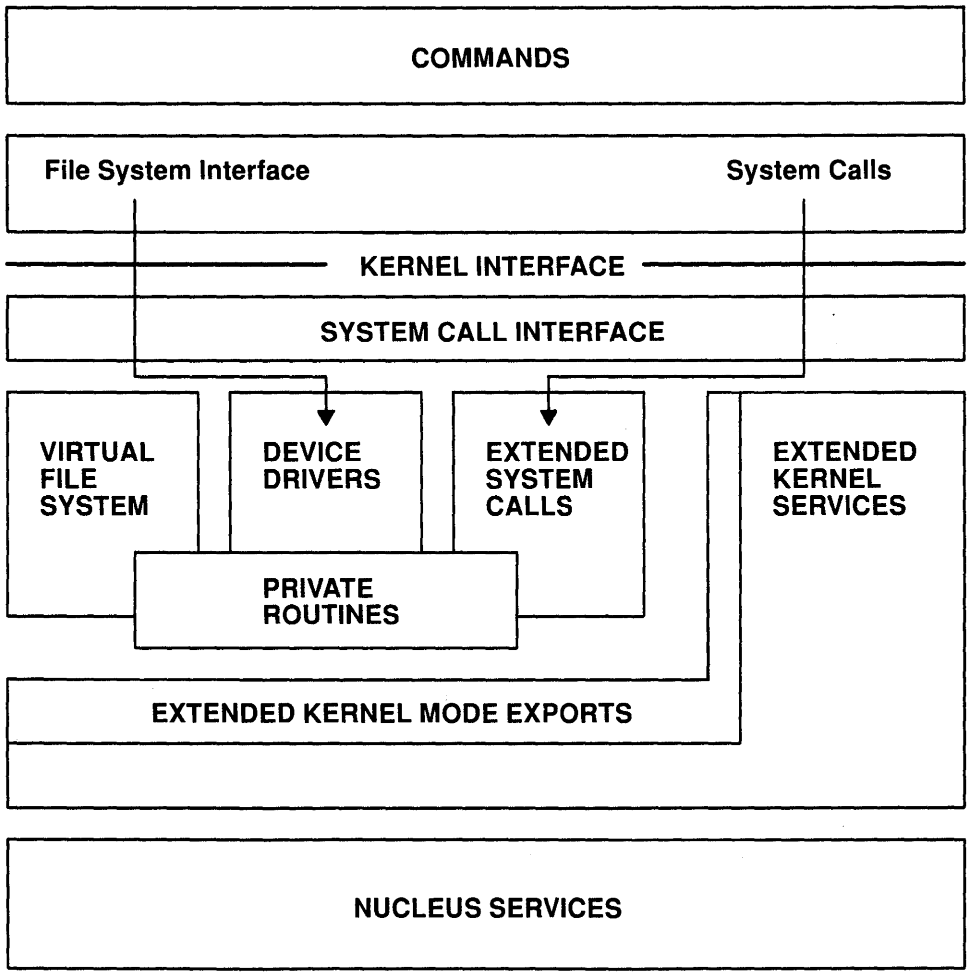
Kernel Environment Programming

The following topics are available as guidance on programming in the kernel environment:

- Kernel Extension Binding
- Execution Environments
- Kernel Processes
- Accessing User-Mode Data while in Kernel Mode
- Understanding Locking
- Signal Handling
- Exception Handling.

Introduction

The AIX kernel is a dynamically extendable kernel that can be expanded by adding device drivers, system calls, kernel services, or private kernel routines. Extensions can be added at system boot or while the system is in operation. The Types of Kernel Extensions diagram illustrates the addition of extensions to the kernel environment.



Types of Kernel Extensions

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 tuneable 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. Binding of kernel extensions can be performed at link-edit, load, or runtime.

Kernel routines execute in a privileged protection domain and can effect the operation and integrity of the whole system.

Kernel Extension Binding

The AIX kernel provides a set of base kernel services to be used by kernel extensions. These services, which are described in the kernel services documentation, are made available to a kernel extension by specifying the kernel export file, **kernex.exp**, as an import file during the link-edit of the kernel extension. (The link-edit operation is performed by using the **ld** command.)

A kernel extension provides additional kernel services and system calls by supplying an export file when it is link-edited. This export file specifies the symbols to be added to the /**unix** name space. Symbols that name system calls to be exported must specify the **SYSCALL** keyword next to the symbol in the export file.

The kernel extension's export file should also have **#!/unix** as its first entry so that the export file may be used by other kernel extensions as an import file. The **#!/unix** as the first entry in an import file specifies that the imported symbols are to come from the /**unix** name space, which is the global kernel name space. This entry is ignored when used in an export file. Thus, the same file can be used both as the export file for the kernel extension that provides the symbols and as the import file for another kernel extension importing one or more of the symbols.

When a new kernel extension is loaded by the **sysconfig** subroutine, any symbols that were defined in the kernel extension's export file at link-edit time are added to the kernel name space, /**unix**. The loader can also load additional object files into the kernel in order to resolve symbols referenced by the new kernel extension. These additional object files will not have their own exported symbols added to the kernel name space, as these exported symbols are only used to resolve references required during the load of the new kernel extension.

In other words, the kernel name space cannot be expanded without the explicit loading of a kernel object file specifying one or more exported symbols. The symbols that are added to the kernel name space are available to any subsequently loaded kernel object file as an imported symbol.

Object files explicitly loaded into the kernel that export symbols into the space are shared by all kernel extensions, in that only one copy of the object file normally exists in the kernel.

Using System Calls

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 can. (The System Calls Available in the Kernel specifies which system calls can be used by either.) User-mode processes in kernel mode can only use system calls that have all parameters passed by value. Kernel routines executing under user-mode processes cannot directly use a system call having reference parameters.

The latter restriction is imposed because when system calls with reference parameters access a caller's data, they are accessing storage across a protection domain. The cross-domain memory services performing these cross-memory operations support kernel processes (kprocs) as if they too were accessing 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.

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

Kernel extensions can bind with a restricted set of base AIX system calls. This is done by specifying the system call export file **syscalls.exp** as an import file when the kernel extension is link-edited. When loading object files into the kernel, the loader is aware that no protection domain switch is required to access system calls from the kernel. It therefore binds the system call imports to the function descriptor that provides direct access to the system call routine. For user-mode programs, the loader binds system call references to a set of function descriptors that invoke the system call handler to effect a switch of protection domain.

Loading System Calls and Kernel Services

Kernel extensions providing new system calls or kernel services should normally place only a single copy of the routine and its static data in the kernel. When this is the case, the SYS_SINGLELOAD option of the **sysconfig** subroutine should be used to load the kernel extension. This option ensures that only a single copy is loaded, since it only loads a new copy if one does not already exist in the kernel. For this type of kernel extension, an updated version of the object file is loaded into the kernel only when the current copy has no users and has been unloaded.

If a kernel extension can support multiple instances of itself (particularly its data), the SYS_KLOAD option of the **sysconfig** subroutine can be used. This option loads a new copy of the object file even when one or more copies are already loaded. When this mechanism is used, currently loaded routines bound to the old copy of the object file continue to use the old copy. Any new routines (loaded after the new copy was loaded) are bound to the most recently loaded copy of the kernel extension.

Unloading System Calls and Kernel Services

Kernel extensions providing new system calls or kernel services can also be unloaded. For each object file loaded, the loader maintains a usage count and a load count. The usage 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 the object file.

When an explicit unload of a kernel extension is requested, the load count is decremented. If the load count and the usage count are both 0, then the object file is unloaded. However if either the load count or usage count is nonzero, the object file is not unloaded. When programs are terminated or killed, the usage counts for kernel extensions that the programs referenced are adjusted. However, no unload of these kernel extensions is performed when the program terminates even if the load and usage counts become zero.

As a result, a situation could exist in which a kernel extension remains loaded, even though its load count has been decremented to 0 (due to unload requests) and its usage count has reached 0 (because of program terminations). In this case, the kernel extension's exported symbols are still available for load-time binding unless another unload request for any object file is received. If an explicit unload request (for any program, shared library, or kernel extension) is received, the loader unloads all object files that have both load and usage counts of 0.

The **slibclean** command, which unloads all object files with load and use counts of 0, can be used to remove object files that are no longer used from both the shared library region and the kernel. Periodically invoking this command reduces the effects of memory fragmentation in the shared library and kernel text regions by removing object files that are no longer required.

Using Private Routines

The previous discussions have been concerned with importing and exporting symbols *from* and *to* the /**unix** common kernel name space. These symbols are global in the kernel and can be referenced by any routine in the kernel.

Kernel extensions can also consist of several separately link-edited object files that are bound at load time. This is particularly useful for device drivers, where one object file contains the top (pageable) half of the driver, while the bottom (pinned) half of the driver is in a second object file. This is also useful where several kernel extensions use common routines provided in a separate object file.

In both cases, the symbols exported by the private object files should not be added to the global kernel name space. If a kernel extension is to have certain symbols exported to the global kernel name space and others used only to resolve references to other private object files, it should be divided into separately link-edited object files. (One object file would contain the symbols to be exported to the kernel name space, while the other would contain the exported symbols that are considered private.)

For object files that reference each other's symbols, each should use the other's export file as its own import file during link-edit. The export file for the object file providing the services should specify **#!** *path/file* as the first entry in the export file, where *path* specifies the directory path to the object *file*, which provides the exported symbols at load time. This entry is ignored when used as an export file. When used as an import file, however, the entry tells the loader where to find the object file resolving the imported symbols at load time.

The object file exporting symbols to the kernel name space should specify **#!/unix** as the first entry in its export file. This allows the export file to be used as an import file by other kernel extensions. The object file containing the symbols to be exported to the kernel name space must be the one explicitly loaded into the kernel with the **sysconfig** subroutine. The loader then loads other private object files, as necessary, to resolve imported symbols required in the load.

When the loader encounters an imported symbol that is resolved by an object file already loaded during the same explicit load request, the loader does not load a new copy. Instead, it resolves the symbol to the copy of the object file already loaded. This allows for cross-resolving symbols between two or more object files loaded as a result of the same explicit load request.

Note: The loader hashes the path and file name of the object file to determine whether the file has already been loaded during this explicit load request. Another copy of the object file could be loaded if differing path names have been used for the same object file and the two names do not hash to the same value.

Object files loaded automatically due to symbol resolution do not have their own exported symbols added to the kernel name space. These symbols remain private to the two or more object files loaded with an explicit load request. In this way, the kernel allows object files to have cross-dependent symbol references, and the loader will correctly resolve them.

Note: When two separate explicit load requests have private symbols resolved by the same object file, two copies of that object file are loaded into the kernel. Each explicit load resolves its symbols to its own private copy of the object file. The private object files can also be combined into libraries with the **ar** archive command.

Using Libraries

A library is a collection of previously link-edited object files or import files and is created by using the **ar** archive command. Each object file or import file within the archive (library) is referred to as a member. AIX program management allows an object file or member to be designated as shared when it is link-edited. Libraries either with or without shared objects can be created and used by kernel extensions. However, library services provided for user-mode applications should not generally be used by kernel extensions, due to the different programming requirements in the kernel.

When the linkage editor (**ld**) resolves a symbol to a library member or object file not designated as shared, it binds the required object file into the output object file in order to resolve the references. However, when symbols are resolved to a library member or object file designated as shared, the shared object file is not included in the output object file. Instead, the linkage editor adds information to the loader section of the output object file. The loader uses this information at load time to learn the location of the shared object file that resolves the symbol.

When these shared object files (normally in libraries) are referenced by user-mode programs, the loader checks the shared library region to determine if the object file is in the shared library region. If it is, the references are resolved to the object file in the shared library region. If the object file has not already been loaded, the loader will load it into the shared library region if the file permissions permit it. In this way, common or shared object files used by user-mode applications can be shared by all user-mode programs in the system.

Unlike user mode, the kernel does not provide a shared library region. Therefore, when a kernel extension that refers to a shared object file is loaded, the loader loads a new copy of the shared object file into the kernel to be used to resolve all references to the object file during the explicit kernel extension load request. However, within the same explicit load request, all references to the *same* object file are resolved to the single copy of the object loaded for the current load request.

AIX provides two libraries that can be used by kernel extensions. The **libcsys** library is a subset of routines found in the user-mode **libc** library that can be used by kernel extensions and consists of the following 25 routines:

- **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**.

The **memccpy**, **memcmp**, **memcpy**, and **memmove** memory routines are low-level routines that the **bcmp**, **bcopy** and **ovbcopy** routines use and can be called directly where path length is critical. These services are documented in the Base Programming documentation as routines found in the **libc** library. The services can be bound to the kernel export by specifying **libcsys.a** as a library in the link-edit of the kernel extension.

The **libsys** library provides a set of kernel services that must be bound with the kernel extension to be used by the extension. These kernel services are documented in the Kernel Services documentation and are described as being **libsys** services in their respective descriptions. The 5 services are:

- **d_align**
- **d_roundup**
- **timeout**
- **timeoutcf**
- **untimeout**.

These services can be bound to the kernel extension by specifying **libsys.a** as an import library in the link-edit of the kernel extension.

Execution Environments

There are two major environments under which a kernel extension can be executed. A kernel extension is said to be executing 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 which environment it is being called in by calling the **getpid** kernel service. This service returns the process identifier of the current process, or a value of -1 if called in the interrupt environment. Some kernel services can be called in both environments, while others can only be called in the process environment.

Process Environment

A routine is said to be executing in the process environment when it is invoked by a user-mode process or by a kernel process. Routines executing in this environment are executed at an interrupt priority of INTBASE. A kernel extension executing in this environment can cause page faults by accessing pageable code or data. It can also be pre-empted by another process of equal or higher process priority.

Routines executing in the process environment can *sleep* or be interrupted by routines executing in the interrupt environment. Kernel routines executed on the behalf of a user-mode process can only invoke system calls that have no parameters passed by reference. Kernel processes, however, can use all system calls in the System Calls Available in the Kernel. Floating-point functions cannot be used in the kernel at all.

Interrupt Environment

A routine executes in the interrupt environment when invoked on behalf of an interrupt handler. A kernel routine executing in this environment cannot cause page faults by accessing pageable code or data. In addition, it has a stack of limited size, is not pre-emptable by another process, and cannot perform any function that would cause it to sleep.

Routines in this environment are only interruptable either by interrupts at a priority more favored than the current priority or by exceptions. These routines may not use system calls and can use only kernel services available in both the process and interrupt environments.

A process in kernel mode may also put *itself* into an environment very similiar 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** 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**, **lockl**, and **unlockl** services can be used if the event word or lock word is pinned or if the process wishes to sleep or use locks.

Note: Locks should only be used when serializing access with respect to other processes. They are not adequate when attempting to serialize access to a resource accessed by a routine executing in the interrupt environment.

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. Documentation on Understanding Interrupts provides more information.

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

Kernel Processes

A kernel process (kproc) is a process that was 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 I/O and device management is required.

A kproc exists only in the kernel protection domain and differs from a user process in the following ways:

- 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.
- 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.
- It is not preemptible by signals.
- Its text and data areas come from the global kernel heap.
- It cannot use shared libraries as such and has no shared library region.
- 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 in order to serialize process-time access to critical data structures.

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 kproc will not have a root directory nor 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 boot or runtime has been reached. This is because the AIX 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

A kernel process can access data that user processes cannot because kernel processes execute in the more privileged kernel protection domain. 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** structure in order to maintain modularity and increase portability of code to other platforms.
- The stack for a kernel process
  The location of the stack for a kernel process is implementation-dependent. This stack can be located in global memory or in the kernel process's process-private segment. A kernel process must not automatically assume 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 a kernel process executes in the kernel protection domain, it 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 dynamically allocated by a kernel process is not automatically freed upon process exit.

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 kernel process must obtain a cross-memory descriptor for the user-mode region to be accessed. This is done by calling the **xmattach** kernel service, which provides a descriptor that can then be made available to the kernel process.

The kernel process should then use 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

Kernel processes (kprocs) are created by a kernel-mode routine by calling the **creatp** kernel service. This service allocates and initializes a process block for the kernel process and sets the new process's state to idle. This new kernel process does not execute until initialized by the **initp** kernel service, which must be called in the same process that created the kernel process (with the **creatp** service). The **creatp** kernel service returns the process identifier for the new kernel process.

Once the **initp** kernel service has completed the kproc's initialization, the kproc is placed on the run queue. On the first dispatch of the newly initialized kernel process, the process begins execution at the entry point previously supplied to the **initp** kernel service with the initialization parameters previously specified on the call to the **initp** kernel service.

A kernel process terminates when it executes a return from its main entry routine. A kernel 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 that called the **creatp** and **initp** kernel services to create the kernel process) receives a *death-of-child* signal. However, it is sometimes undesirable for the parent process to receive this death-of-child signal due to kproc termination. In this case, the kproc can call the **setpinit** kernel service to redesignate the **init** process as its parent. The **init** process cleans up the state of all its children processes that have become zombies. A kernel process can also issue the **setsid** subroutine call to change its session so that signals and job control affecting the parent process's session do not affect the kernel process.

Kernel Process Pre-emption

A kernel process is initially created with the same process priority as its parent. It can therefore be pre-empted 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** or **nice** subroutines to modify their execution priority.

The kernel process can use the kernel locking facilities (the **lockl** and **unlockl** kernel services) to serialize access to critical data structures. This use of locks does not guarantee that the process will not be pre-empted, but it does insure that other processes trying to acquire the lock will wait 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 the interrupt-control facilities in the kernel (such as the **i_disable** and **i_enable** kernel services). Kernel processes must ensure that no access to pageable code, data, or stack is made while executing at an interrupt priority higher than INTBASE. Kernel processes are not pre-empted by other processes while executing at an interrupt priority higher than INTBASE. However, they can still be interrupted by interrupts that are more favored than the current interrupt priority level.

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

Kernel processes, unlike user processes, are not pre-emptible by signals, even the **SIGKILL** signal. Kernel processes must *poll* for signals in order for them signals to be delivered. Polling ensures the proper kernel-mode serialization, since signals to user-mode processes are not delivered while in kernel mode, and kernel processes are always in kernel mode.

Signals that have action applied at generation time (rather than delivery time) have the same effect regardless of whether the target is a kernel or a user process. Kernel processes can poll for unmasked signals 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 invoke signal handlers for kernel processes as it does for user processes.

Kernel processes should also use the exception-catching facilities available in kernel mode to handle exceptions that can be caused during execution 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 may possibly occur during execution. Refer to Understanding Exception Handling 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 since 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 services recognize that the system call function is executing within a kernel process instead of a user process and correctly handle the data accesses.

However, the error information returned from a system call made by a kernel process must be accessed differently than for a user process. A kproc 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 callers.

Kernel processes can use only a restricted set of the base system calls found in the **syscalls.exp** export file. System calls available to kernel processes can be found in the List of System Calls Available in Kernel Mode.

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 the data access. These services also prevent system crashes in a system call when accessing user-mode data. These services can only be called when executing in the process environment of the process containing the user-mode data.

User-mode data access primitives are:

- The **subyte** and **suword** services store either a byte or a word to user memory.
- The **fubyte** and **fuword** services fetch either a byte or a word from user memory.
- The **copyin** or **copyout** services copy data between user and kernel memory.
- The **copyinstr** service copies a null-terminated character string from the user-mode address space into kernel memory. The copy is halted after the first null character is encountered.

An addiitional set of services allow data transfer between user mode and kernel mode when a **uio** stucture is used. (This structure describes the user-mode data area to be accessed.) These services, typically used between the file system and device drivers to perform device I/O, are the following:

- The **uiomove** service
- The **ureadc** and **uwritec** services.

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 in an asynchronous fashion. Examples of this 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. This service must be called·in the process environment of the process containing the data area.

Once a cross-memory descriptor has been obtained, the **xmemin** and **xmemout** kernel services can be used to access the data area outside of the process environment containing the data. As soon as 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 required. Their use increases the difficulty of porting the kernel extension to other machine platforms because of the machine dependencies of cross-memory operations.

Understanding Locking

A *conventional lock* is used to serialize access to a predefined data structure. It is conventional in that all users of the data structure must lock the data structure's conventional lock before accessing the data structure. When finished, the users must also unlock the data structure's conventional lock.

A conventional lock has two states: locked or unlocked. In the *locked* state, a process is currently accessing the data structure associated with the conventional lock. This process is referred to as the owner of the conventional lock. No other process that attempts to lock the conventional lock can get the lock until the process that owns the conventional lock unlocks it with the **unlockl** kernel service. In the *unlocked* state, no process accesses the data structure or owns the conventional lock.

When a lower priority process owns a lock that a higher priority process is attempting to acquire, the priority of the process owning the lock is raised to the process priority of the highest priority process waiting to acquire the lock. When the process with boosted priority releases the lock, the priority of that process is restored to its normal value.

Locking Strategy in Kernel Mode

A linear hierarchy of locks exists, within which the global kernel lock, **kernel_lock**, has the the coarsest granularity. A kernel extension should not attempt to acquire the kernel lock if it owns any other lock. This hierarchy is imposed by software convention and is not enforced. The ordering of locks follows:

- The **kernel_lock** global kernel lock
- File system locks (private to file systems)
- Device driver locks (private to device drivers)
- Private fine-granularity locks.

Locks should be unlocked in the reverse order in which they were acquired.

Signal Handling

For information on signal handling for a user-mode process in kernel mode, see Handling Signals While in a System Call on page 4–4.

For information on signal handling while in a kernel process, see Kernel Process Signal and Exception Handling on page 1–9.

Exception Handling

There is 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 occurred.

- An exception is a synchronous event and is directly due to 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 its state and forces a branch to a particular location. On decoding the reason for the machine interrupt, the interrupt handler determines whether the event is an interrupt or an exception and performs different processing accordingly.

Note: Ordinary page faults are treated more like interrupts than exceptions. The only difference between a page-fault interrupt and other interrupts is that the interrupted program is made nondispatchable until the page fault is resolved.

Exception Processing

When an exception occurs, the current instruction stream cannot continue. It is almost never appropriate to simply ignore the exception. At the very least, the results of executing the instruction are undefined and thus further execution of the program is effectively meaningless. The AIX kernel provides an exception-handling mechanism by which an executing 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 user mode or kernel mode.

Default Action

If no exception handler is currently defined when an exception occurs, then one of two things usually 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 the AIX kernel mode extends the UNIX **setjump/longjump** context-save-and-restore mechanism by providing **setjmpx** and **longjmpx** kernel services to handle exceptions. The traditional UNIX 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** *service.* When execution returns to this point, the return code from **setjmpx** 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's first level exception handler gets control. The first level exception 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 performs the necessary re-enabling of the capability to perform programmed I/O operations.

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

The **longjmpx** 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** 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 Actions

A typical exception handler should do the following:

- Perform any necessary clean-up, such as freeing storage or segment registers and releasing other resources.
- If the exception is recognized by the current handler and can be handled entirely within this routine, the handler should re-establish itself by calling the **setjmpx** service. This allows normal main-line processing to continue.
- If the exception is not recognized by the current handler, it must be passed along to the previously stacked exception handler. The exception is passed along by calling the **longjmpx** service, which either invokes the previous handler (if any) or takes the system's default action.
- If the exception is recognized by the current handler but cannot be handled, it is treated as though it could not be recognized at all. The **longjmpx** service is called, which either passes the exception along to the previous handler (if any) or takes the system's default action.

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** service) before returning to its caller. Note that when the **longjmpx** kernel service invokes an exception handler, that handler's entry is automatically removed from the stack.

Implementing Kernel Exception Handlers

The **setjmpx** kernel service provides a means of saving the following portions of a program's 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 at a later point by calling the **longjmpx** service, which accomplishes the following:

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

The **setjmpx** 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 filling in the state data in the jump buffer, the **setjmpx** kernel service pushes the buffer onto the top of a stack maintained in the machine state save structure.

The **longjmpx** service is used to return to an earlier point in the code, namely the point at which the **setjmpx** service was called. Specifically, the **longjmpx** 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 argument to the **longjmpx** service is an exception code that is passed to the resumed program as the return code from the **setjmp** service. The resumed program tests this code to determine the conditions under which the **setjmpx** service is returning. If the **setjmpx** service has just saved its jump buffer, the return code is a value of 0. On the other hand, if an exception has occurred, then the program has been re-entered by a call to the **longjmpx** service, which has passed along a return code not equal to 0.

Note that only the resources listed above are saved by the **setjmpx** service and restored by the **longjmpx** service. Other resources, in particular segment registers, are not restored. A call to the **longjmpx** service, by definition, returns to an earlier point in the program. It is the program's responsibility to free any resources that may have been allocated between the call to the **setjmpx** service and the call to the **longjmpx** service.

If the exception handler stack is empty when the **longjmpx** service is issued, there is no place to jump to and the kernel's default action is taken. If the stack is non-empty, then the context defined by the topmost jump buffer is reloaded and resumed. The topmost buffer is removed from the stack.

The **clrjmpx** service removes the top element from the stack as placed there by the **setjmpx** service. The caller to the **clrjmpx** service is expected to know exactly which jump buffer is being removed, as this should have been established earlier in the code by a call to the **setjmpx** service. Accordingly, the address of the buffer is required as a parameter to the **clrjmpx** service so that it can perform consistency checking by **ASSERT**ing 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 **setjmpx** call 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** 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** 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** service is nonzero), the jump buffer is automatically removed from the list of jump buffers. In this case, a call to the **clrjmpx** 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. Alternatively, if the last value of the variable is desired at exception time, the variable 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 **<sys/except.h>** header 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** 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 **malloc** kernel service).
- Call the **longjmpx** 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 does recognize 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** service to establish an exception handler) and be assured that the resources will later be released. This is because no matter what events occur, the exception handler gets a chance to release those resources 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). In order to implement this scheme, the set of exception codes used cannot have duplicates.

Exceptions generated by hardware use one of the codes in the **<except.h>** header file. However, the **longjmpx** service can be invoked by any kernel component, and any integer can serve as the exception code. Thus a mechanism similar to the old-style **setjmp** and **longjmp** 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 be guaranteed not to 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 (that is, by using its function descriptor). Later on in the calling sequence, after any number of intervening calls to the **setjmpx** service by other programs, a program can issue a **longjmpx** call 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** service again.

Addresses of function descriptors are not the only possibilities for unique code numbers. For example, addresses of external variables can also be used. By using addresses that are resolved to unique values by the binder and loader, the problem of code space collision is tranformed into a problem of external name collision. This problem is not only much more easily solved, but is also routinely solved whenever the system is built. By comparison, preassigning exception numbers by using **#define** statements in a header file is much more cumbersome and error-prone.

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** 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 systemwide 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.

Related Information

The **setjmpx** kernel service, **longjmpx** kernel service, **clrjmpx** kernel service, **getexcept** kernel service, **malloc** kernel service, **uexadd** kernel service and **uexdel** kernel service.
Handling Signals While in a System Call on page 4–4.
Writing a Device Driver on page 2–1 .
Extending the Kernel with Device Drivers on page 3–1 .
Extending the Kernel with System Callson page 4–1 .
Kernel Services on page 6–1.
Alphabetical List of Kernel Services on page A–1 .
Using the Kernel Debugger in *General Programming Concepts*.

Writing a Device Driver

The following topics are provided as guidance for understanding how a device driver is organized and how it fits into the operating system environment:

- Concepts Overview
- Device Driver Classes
- Device Driver Roles
- Device Driver Structure
- Understanding I/O Access Through Special Files
- Understanding the Device Switch Table
- Understanding Major and Minor Numbers for a Special File.

Concepts Overview

Device drivers are kernel extensions that control and manage specific devices used by the operating system. The I/O subsystem, in conjunction with the device drivers, allows processes to communicate with peripheral devices such as terminals, printers, disks, tape units, networks. Device drivers may be installed into the kernel to support a class of devices (such as disks) or a particular type of device (such as a specific disk drive model). Device drivers shield the operating system from device-specific details and provide a common I/O model for accessing the devices for which they provide support.

The operating system also supports and uses the concept of pseudo-devices. Understanding Pseudo-Device Drivers on page 3–10 provides more information.

The system interface to devices, which is supported by device drivers, is through the file system. Each device that is accessible to a user-mode application has a file name and can be accessed as if it were an ordinary file. By convention, this device file name is found in the /**dev** directory in the root file system. This device file name along with its associated inode is known as a device special file.

Conceptual Organization of Device Drivers

Device drivers may be characterized by the *class* of I/O subsystem interfaces that they provide. Device driver routines may be further characterized by the *roles* that they play in supporting the device I/O. Finally, the overall *structure* of the device driver is dictated by the execution environment in which the routines execute.

Device Driver Classes

The AIX operating system supports two classes of device drivers: *character* and *block*. These classes of device driver are distinguished by the types of devices they support and the interfaces that are presented to the kernel.

The block device interface is suitable for random access storage devices with fixed-size addressable data blocks. Devices supported by block device drivers can also potentially support a mounted file system. User-mode access to these block device drivers is through a block device special file.

The character device interface is more suitable to other types of devices (such as terminals, printers, and networks) that do not have strict definitions of fixed-size addressable data blocks. These devices cannot directly support mounted file systems. User-mode access to these character device drivers is through a character device special file. Character device drivers are not as highly structured as block device drivers.

Block device drivers can provide character device interfaces and access to their block devices by providing a character special file, as well as the block special file. Character device access to block devices is called *raw I/O*.

Comparison of Block and Character Device Drivers

Device drivers of both classes have entry points registered in the device switch table. Character device drivers typically have read and write entry points defined in the device switch table for providing data transfer operations. Block device drivers on the other hand have a strategy entry point instead. Block device drivers providing raw I/O access typically provide read, write, and strategy entry points for providing data transfer operations.

A major difference between block and character device support is in how their read and write I/O requests are processed. Both types have entry points registered in the device switch table and both perform I/O as a result of file system calls to their devices. However, read and write requests directed to a block device are managed by a kernel buffering mechanism not present on requests directed to character devices.

Read and write I/O access using character special files invokes corresponding entry points in a character device driver. The same calls using block special files, however, do not directly invoke corresponding block driver entry points. Instead, the file system processing these calls invokes buffer management facilities provided by the kernel. These buffer management routines then call the block device driver strategy routine (**ddstrategy**) when required. The buffer management facilities determine when a data transfer is required and call the correct block driver strategy routine at that time.

Device Driver Roles

Device drivers can play two roles in the AIX operating system: the *device head* role and the *device handler* role. Character and block device drivers can provide one or both roles. A particular entry point for a device driver is either a device head or a device handler entry point.

Device Head

Device driver routines performing the *device head* role are responsible for fielding the device driver request generated by a user application. Such requests are submitted through the use of file system calls or possibly by another kernel extension using the kernel file system services.

Device head routines have their entry points installed in the device switch table. Examples of these routines are the device driver **ddconfig**, **ddopen**, **ddclose**, **ddread**, **ddwrite**, **ddioctl**, **ddmpx**, and **ddrevoke** routines. User applications can use file system calls in conjunction with special files to access these routines, while kernel extensions can use the file system services available in the kernel (the Logical File System **fp_xxxx** services).

Device head routines are responsible for the following functions:

- They convert the request from the form of the file I/O function call to a form that the routines acting in the corresponding device handler role understand.
- They perform the appropriate data blocking and buffering.
- They manage the device. This includes such actions as maintaining queues of I/O requests and handling error recovery and error logging.

Routines providing the device head role must conform to the programming model described in System Call Kernel Extension Overview because they are called by system calls and execute in the same environment.

Device Handler

Device driver routines fulfilling the *device handler* role perform the actual I/O to and from the device. User applications cannot directly access these routines without going through the device head routines. Examples of device handler routines are the **ddstrategy** and **dddump** device driver entry points, the interrupt handler, start I/O, and I/O exception handling routines.

Support for some devices can be implemented using two separate device drivers. The first driver acts in the device head role while the second mainly performs the device handler role. However, this second (device handler) driver can also have its own set of small device head routines, and these routines would be registered in the device switch table. The extra set of device head routines is provided primarily to make system configuration and binding easier. A device driver of this type can be completely inaccessible to application programs or can provide a special file for diagnostic purposes.

Device Driver Structure

Device driver routines providing support for physical devices typically execute in two different types of environment, thus leading to a two-part structure. One part, referred to as the top half of the device driver, always executes in the process environment. Routines in this part typically provide the device head role, because they are always executed in the environment of the calling process.

The other part, referred to as the bottom half of the device driver, executes in the process or interrupt environment. Routines in this part normally provide the device-handling role because they deal with actual device I/O typically driven by hardware interrupts. Additionally, for block devices, the strategy routine is found in the bottom half because it may be called in the interrupt environment due to paging or other asynchronous requests.

Device Driver Top Half Routines

Because routines in the top half of a device driver are only called in the process environment, the code and data accessed in this environment are normally pageable. The AIX kernel is designed to allow large portions of kernel code and data to be pageable in order to decrease the amount of physical memory required by the kernel. This is very important for the AIX kernel because the design philosophy is to create fairly large data structures in pageable virtual memory. These large data structures can then support a wide range of system loads and configurations.

Preemption in the AIX Operating System

The AIX kernel is designed to allow preemption by other processes while executing in kernel mode. This change to allow preemption was made in order to enhance support for real-time processes and large multiuser systems.

Most existing UNIX device drivers do not expect this form of preemption. The effects of preemption on the existing UNIX device drivers can be minimized by serializing the execution of these types of device drivers. This can be done by using the **unlockl** kernel service with the **KERNEL_MODE** lock. This does not disable preemption of the device driver, but ensures that all device drivers of this type are serialized with respect to each other. Understanding Locking provides more information on using locks.

Device Driver Bottom Half Routines

The second half of the device driver structure is referred to as the *bottom half.* This half of the device driver typically consists of a routine that starts I/O operations (start I/O), an interrupt handler, and (optionally) off-level interrupt handling and device time-out routines. The device driver's strategy and dump routines are also considered part of the bottom half.

The start I/O routine is typically known only to other routines within the device driver, such as the strategy and interrupt-handling routines. Interrupt handling routines are registered using kernel services. The dump and strategy routines (the **dddump** and **ddstrategy** entry points) are found in the device switch table.

Some character device drivers, particularly pseudo-device drivers, may have no bottom half if they have no need to execute in the interrupt environment.

This part of the device driver executes in both the interrupt handler environment and in the environment of the calling process. Both the code for this part of the device driver and the data it accesses must be pinned so that page faults are not taken in the interrupt execution environment. In addition, routines in the bottom half can use only kernel services that are specified as callable in the interrupt environment.

Serialization and Preemption in the Bottom Half of the Device Driver

Execution serialization in bottom half routines is accomplished by using interrupt control functions. Unlike top half routines, bottom half routines may not use locks for serialization because their use may cause a page fault or an attempt to sleep while executing in the interrupt environment.

The interrupt control functions provided by the kernel provide serialization by allowing a routine to mask interrupt levels or disable interrupt priorities. These can be used by the bottom half routine to prevent it from being interrupted by other routines for which serialization is required. The kernel also provides associated interrupt level unmasking and priority-enabling functions to resume previously disabled interrupt handling, once the critical serialization section has been executed. Bottom half routines using these services should only disable interrupts to the least favored priority that still provides the necessary serialization.

Bottom half routines may also be executed in the process environment, which is preemptible. The interrupt priority control functions used in the interrupt environment also provide the necessary serialization when used in the process environment. This is due to the fact that dispatching and process preemption are only performed at the least favored interrupt priority, called INTBASE. The interrupt-level masking services do not provide process serialization and should be used only when serialization is required in the interrupt environment.

Understanding I/O Access Through Special Files

The kernel contains many entry points into the file I/O subsystem. Common entry points are invoked using the **open**, **close**, **read**, **write**, **lseek**, and **ioctl** subroutines. The file I/O subsystem determines whether the request is to gain access to an ordinary file, a block special file, or a character special file. In the case of device special files, this subsystem translates the file name into a major and minor number, which are used to select the device driver and specific device.

Warning: Potential for data corruption or system crashes: Data corruption, loss of data, or loss of system integrity will occur if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Block special files are provided for logical volumes and disk devices on AIX and are solely for system use in managing file systems, paging devices, and logical volumes. They should not generally be used for other purposes.

Access to Character Device Drivers

Character device drivers may only be accessed by performing file I/O associated with a character special file. When processing an open or create request associated with a character special file, the system always calls the device driver's **ddopen** entry point to allow any special processing to occur (for example, device initialization or resource allocation). The device driver's **ddclose** entry point, however, is normally only called when the last process having the special file open closes it.

When a read or write associated with a character special file occurs, the file system constructs a **uio** structure containing the user's arguments and file-table data to be passed to the device driver's **ddread** or **ddwrite** entry points. This **uio** structure describes:

- Address of the user's buffer
- Data transfer count
- Current device data offset obtained from the file table entry
- Current open mode entry or file control information obtained from the file table.

This **uio** structure is then passed to the **ddread** or **ddwrite** entry point of the character device driver, which performs the data transfer. As a result of this transfer, the fields in the **uio** structure are updated to reflect the amount of data actually transferred and the new device data offset.

Access to Multiplexed Character Device Drivers

Access to multiplexed character device drivers is similar to standard character device drivers, except that the concept of channels has been added. A channel is typically supported by a device driver as a resource subunit on a particular device. Each subunit can be selected by an extra suffix on the special file path name.

As explained previously, the particular device is accessed using a character device special file containing a device major and minor number. When an open or create request is made involving a multiplexed character special file, the path name of the special file can be followed by a character string specifying the name of the channel being requested. If no name is provided when opening a multiplexed character driver, the device driver typically assigns the next available channel.

For example, the special file for the high function terminal device is named **/dev/hft**. Virtual terminals on that physical terminal are assigned channels. If an open of **/dev/hft** is specified, the multiplexed device driver assigns the next available virtual terminal. However, if an open of **/dev/hft/***n* is specified, a specific virtual terminal is being requested and is opened by the device driver.

Character device drivers may be supported as multiplexed if they provide and register a **ddmpx** routine in the device switch table. When processing an open or create request associated with a character special file, the system always determines if the associated device driver has a **ddmpx** routine specified in the device switch table. If it does not, standard character device open processing occurs.

If a **ddmpx** routine is found, the system calls the device driver's **ddmpx** routine and passes it a pointer to a character string specified after the special file name. If the character device driver can successfully allocate a channel, it returns a channel ID to the system. The system then calls the device driver's **ddopen** routine with the channel ID received from the **ddmpx** routine to allow for any special processing such as device initialization or resource allocation. This channel ID accompanies file I/O requests associated with the particular open or create call that assigned it.

Unlike a standard character device driver, a multiplexed driver's **ddclose** routine is called once for every close that had an associated open or create request. Once the file system has determined that the last close has been issued for a channel, the multiplexed driver's **ddmpx** routine is called with an indication that the channel should be deallocated.

For a multiplexed device driver, a count of the number of explicit opens can be maintained. However, a count of the number of using processes (due to **fork** and **dup** subroutines) cannot. Keeping a count should be required only in unusual circumstances, because the last close for a channel can be recognized by the channel deallocation call to the **ddmpx** routine.

Channels offer the advantage of allowing access to a very large number of dynamically allocated subunits without the need for a large number of special files. The availability of channels may also be allowed to shrink or grow dynamically as the availability of resources changes. Once a channel has been opened, its permissions and other security attributes can be changed independently of other channels or the base special file.

Access to Block Device Drivers

When processing an open request associated with a block device driver, the system always calls the device driver's open routine to allow any special processing (such as device initialization or resource allocation) to occur. However, the device driver's close routine is called only when the last process having it open closes the device.

Read and write requests to block device drivers are handled by several different mechanisms in the AIX Base Operating System. In the following discussion, the first three mechanisms all use the block interface of the device driver for data transfers and specify the data transfer parameters in a **buf** structure passed to the strategy routine.

Access to Block Devices Designated as Paging Devices

The first mechanism involves a block device designated as an active paging device. In this situation, the pager invokes the device driver **ddstrategy** entry point for page-out or page-in data transfers. The pager supplies the necessary **buf** structures from its own pool and the associated data buffers are memory pages.

Access to Block Devices By the File System and Virtual Memory Manager

Secondly, block device drivers can be accessed by the file system and virtual memory manager. This may be due to user file I/O or to access of file system meta-data (internal file system data). In this case, the file system, the virtual memory manager, and the pager cooperate to provide buffer caching mechanisms using the underlying memory pages, instead of providing a caching layer on top of page management.

File I/O Access to Block Devices Using the Block Special File

A third mechanism for access to block device drivers is through file I/O using the block device special file. If read and write requests are directed to a device using a block special file, the requests are managed by the block I/O buffer cache mechanism in the kernel. This buffer cache mechanism attempts to increase efficiency by keeping in-memory copies of frequently used blocks of data, thus reducing physical I/O requests to the device. This buffer cache mechanism also provides multiple processes with a consistent view of the data in the block. This is true because when separate processes request I/O to the same block on the same I/O device, they all access the same buffer in the cache.

Potential Hazards of Block Special File Usage

Data corruption or loss of data and system integrity occur if devices supporting paging, logical volumes, or mounted file systems are accessed using block special files. Unlike previous UNIX operating systems, AIX file systems and paging support use the physical memory page management functions in the virtual memory manager to perform the buffer caching traditionally supported by the block I/O buffer cache management.

The block I/O buffer cache is still used, however, for buffer caching of operations directed to block special files. Accessing these devices through the block special files can result in more than one copy of the data existing in memory. The virtual memory manager can be maintaining one copy while the block I/O buffer cache contains a second. This can lead to potentially disastrous results.

When block devices are not used by the paging subsystem or file systems, all direct data accesses transferring large blocks of data should use the raw I/O character special file for best performance. Using the block I/O buffer cache provides poorer performance due to the small buffer cache size and the non-optimum scheduling of I/O to the device by the block I/O buffer management support. The block I/O special files should only be used to perform I/O when small unaligned or odd-sized data transfers are being requested and performance is not a concern.

Block special files are provided for logical volumes and disk devices on AIX solely for system use in managing file systems, paging devices, and logical volumes. They should not generally be used for other purposes.

Raw I/O Access to Block Devices Using the Character Special File

A fourth way to access some block device drivers is through raw I/O using a character special file name. If a block device driver supports this type of access, the file system invokes the device driver's read and write routines for file I/O associated with the character special file. (Raw I/O access is supported by providing a character special file and read/write entry points in the device switch table.)

To the file system, this mechanism is identical to character device driver access mechanisms. The block device driver however must provide raw I/O read and write processing routines. Because the block device driver converts the raw I/O requests to block requests, raw I/O typically has data transfer restrictions associated with the driver and device that are not normally found with ordinary character device I/O. For example, typical restrictions are that data transfers must be in multiples of the block size or that data transfers must start on a block boundary.

Understanding the Device Switch Table

The file system accesses character or block device driver routines through a table called the *device switch table.* This table is kept in kernel storage and contains one element for each configured device driver. Each element is itself a table of entry point addresses. There is one address for each entry point provided by that device driver.

A device driver's entry points are inserted in the device switch table at device driver configuration time. The driver's configuration routines call various kernel services to install driver entry points into one or more entries (rows) of the table. Each table entry or row is indexed by a major number.

Understanding Major and Minor Numbers for a Special File

Major numbers are assigned at device configuration time by the configuration management routines used by device configuration methods (in particular, the **genmajor** configuration library routine). The major number assigned to a device driver for its entry into the device switch table is the same as the major number in the device special file associated with the device.

Devices are generally identified in the kernel through major and minor numbers. Usually, a major number identifies a particular device driver. Minor numbers identify various device instances known to the device driver. However, a device driver may be assigned multiple major numbers. Also, minor numbers can be used to identify different modes of operation for a device as well as different device instances.

Programs do not need to understand these major and minor numbers to access devices. A program accesses a device as though it were a file by opening the device's corresponding special file located in the **/dev** directory. The special file's inode contains a particular major and minor number combination specified when the special file was created. This relationship remains constant until the special file is deleted.

The major number uniquely identifies the relevant device driver and thus is used to index into the device switch table maintained by the kernel. The interpretation of the minor number is entirely dependent on the particular device driver. Most frequently, the minor number is used to select one of multiple subdevices supported by the device driver. As a minor device number, it usually serves as an index into a device driver-maintained array of information about each of several devices or subdevices supported by the device driver.

Creation of Major Numbers

The first time a device is configured, its Configure method is responsible for determining the major and minor numbers for the device and for creating the device's special files. When subsequently configured, the device's Configure method must ensure that the same major and minor numbers are used to describe the device to the device driver. This consistency guarantees that the previously created special file allows access to the same device as it did previously.

Major numbers are allocated to device driver instances. When the **genmajor** routine is invoked with a particular device driver instance name passed as a parameter, it will:

- Return the major number corresponding to the device driver instance name, if it has already been allocated, or
- Assign the next available major number to the specified device driver instance and return the newly assigned number.

Each time a device is configured, its Configure method should simply call the **genmajor** routine with the device's device driver instance name. If the device has not been assigned a major number, the **genmajor** routine assigns one and returns it. Otherwise it returns the previously assigned number.

A device's device driver instance name is obtained from the Device Driver Instance descriptor of the device's CuDv object. This descriptor is usually filled in by the Define method when the device is first customized. For most devices, the device driver instance name is simply the device driver name. If the device driver for a device uses multiple major numbers, a different device driver instance name must be assigned for each major number.

Creation of Minor Numbers

The allocation of device minor numbers is highly device-specific. A device's Configure method can determine minor number assignments on its own or it can use the **genminor** and **getminor** routines. When the **genminor** routine is used to allocate minor numbers for a device, information is stored in the Configuration database, which keeps track of what minor numbers have been assigned for a particular major number, as well as the minor numbers being assigned to the device. The **getminor** routine can be used to obtain a list of minor numbers that have been assigned to a device.

Releasing Major and Minor Numbers

When a device is unconfigured, its special files and major and minor number assignments typically remain intact. The Unconfigure method does not deallocate the assignments or remove special files. This eliminates the need to reassign new values and rebuild special files when the device is once again configured.

The major and minor numbers are to be unassigned when the device is undefined. The Undefine method will also delete the device's special files. If the device's minor numbers were allocated with the **genminor** routine, the **reldevno** routine can be used to both delete the major and minor number assignments and to delete the special files.

Related Information

The inode File.

The **pin** kernel service, **pincode** kernel service **unpin** kernel service, **lockl** kernel service, **unlockl** kernel service.

The **genmajor** configuration subroutine, **genminor** configuration subroutine, **getminor** configuration subroutine, **reldevno** configuration subroutine.

The **ddconfig** device driver entry point, **ddopen** device driver entry point, **ddclose** device driver entry point, **ddioctl** device driver entry point, **dddump** device driver entry point, **ddread** device driver entry point, **ddwrite** device driver entry point, **ddselect** device driver entry point, **ddmpx** device driver entry point, **ddrevoke** device driver entry point, **ddstrategy** device driver entry point.

The Device Configuration Subroutines.

Special File Overview in *General Programming Concepts*

Interrupt Management Services on page 6–7, Understanding Execution Environments on page 1–6 .

Kernel Environment Programming on page 1–1.

Virtual File System Overview on page 5–1, Writing System Calls on page 4–1, Extending the Kernel with Device Drivers on page 3–1, Block I/O Buffer Cache Kernel Services: Overview on page 6–8.

Writing a Device Method on page 7–7.

The Configuration Subsystem on page 7–1.

Extending the Kernel with Device Drivers

The following topics are provided as guidance for extending the kernel with device drivers:

- Understanding Block I/O Device Drivers
- Block I/O Processing
- Understanding Character I/O Device Drivers
- Understanding Off-Level Processing
- Understanding Pseudo-Device Drivers
- I/O Exception Handling Overview
- Interfacing to the Hardware
- Installing and Configuring Device Drivers.

Writing device drivers to extend the kernel has the following advantages over adding system calls:

- Applications access device drivers through the file I/O subsystem. This subsystem provides a uniform security mechanism for controlling access to objects.
- The file I/O subsystem presents a common set of interfaces for accessing the devices. These interfaces provide a degree of device independence at the application level.
- The open and close file processing required by the file I/O subsystem also allows device drivers to maintain per-process information easily.

A disadvantage of device drivers over system calls is that they must conform to the interfaces enforced by the file I/O subsystem, while system calls do not.

Understanding Block I/O Device Drivers

A device driver in the *block class* supports asynchronous I/O transfers in fixed-size blocks, as requested by the operating system. Block device drivers may be used by the operating system's block I/O buffer cache routines, the pager, file systems, and other device drivers.

Block I/O Device Driver Entry Points

The device switch table contains the entry point addresses of the interface routines for each block device driver in the system, just as it does for character device drivers. Like character device drivers, block device drivers supply both a config routine for configuration support (the **ddconfig** device driver entry point) as well as open and close routines (**ddopen** and **ddclose**) called on each open and on the final close of a device. Instead of having separate read and write routines, as character device drivers do, each block device driver has a strategy routine (**ddstrategy**). This routine is called with a pointer to a buffer header, known as the **buf** structure, which contains the I/O request parameters.

Block device drivers can also provide an ioctl routine (**ddioctl**), which is called when an **ioctl** subroutine operation is directed to the device. However, the support of **ioctl** subroutine operations by a block device driver is more restrictive than that of character device drivers. An **ioctl** subroutine operation to a block device must not be required as a prelude to strategy-routine processing of requests. Therefore, **ioctl** subroutine operations provided by a block device driver should only provide optional control functions and must not be required in order for the strategy routine to perform operations to or from the device. This is because **ioctl** operations are typically device-dependent, and the block device interface must be supported as a device-independent interface because it is being used as a generic interface by the base operating system.

Providing Raw I/O in a Block I/O Device Driver

Block device drivers supporting raw I/O can also provide read and write routines (**ddread** and **ddwrite**) with entry points in the device switch table. These routines can be used to provide a more character-like interface to the device. However, they do not provide the full I/O access capabilities provided by many character device drivers. Data blocking restrictions not normally found with character device driver access are typically required for the character read and write interface of block device drivers providing raw I/O.

There are also other differences between character drivers and block drivers providing raw I/O. Asynchronous open, read, and write requests are not normally supported by block device drivers in the way that character device drivers may support them. In addition, device event-notification functions provided by the **poll** and **select** subroutines are not provided by block device drivers. Finally, there is no multiplexed capability available to block device drivers as there is for character device drivers.

Understanding Raw I/O Access to Block Device Drivers on page 3–4provides more information about this dual interface to block devices. Understanding Raw I/O Support on page 3–5 explains how raw I/O requests are processed.

Optional System Dump Support

Block device drivers may also optionally support their device as a candidate target for system memory dumps. The **dddump** entry point is provided in the device switch table for this purpose. In the unlikely event of a system crash, the AIX kernel will initiate a system dump request to a predesignated dump device. Because normal system processing and resources should not be relied upon in this situation, the device driver's dump routine must provide special system dump support to the device.

Unsupported Entry Points

A block driver does not support all of the entry points found in the device switch table because the table is used by both block and character device drivers. If the routine is not provided and should not generate an error when called, the corresponding entry point in the device switch table should specify the **nulldev** entry point. If the call should result in an error return, the **nodev** entry point should be specified. These default routines are provided as part of the base kernel.

Examples of device switch entry points that should be set to the **nodev** base kernel routine are the **ddmpx**, **ddrevoke**, and **ddselect** entry points. If raw I/O is not supported, then the **ddread** and **ddwrite** entry points should also be set to the **nodev** entry point in the device switch table. If the driver does not provide **ioctl** or system dump support, the corresponding **ddioctl** and **dddump** entry points should also be set to the **nodev** entry point.

Block I/O Processing

A discussion of block I/O processing encompasses the following topics:

- Accepting the Request
- Providing notification of I/O completion
- reordering of I/O requests
- Handling out of range block numbers
- Queuing the request to the start I/O Routine
- Starting processing with the start I/O Routine.

Accepting the Request

When the strategy routine (the **ddstrategy** device driver entry point) is invoked, a pointer to a buffer header (or chain of buffer headers) is used as a parameter for requesting device I/O. The buffer header is in the format of a **buf** structure. The role of the strategy routine is to perform the operation as requested by the information in the buffer header or chain of buffer headers. The buffer header contains the following information:

- Major and minor number of the device for which the I/O is intended
- Description of the memory buffer to be used in the data transfer
- Direction of the transfer
- Count of the amount of data to be transferred
- Block number on the device for which the I/O is targeted
- Operation flags.

The strategy routine returns to the caller as soon as the **buf** headers have been queued to the appropriate device queue. The strategy routine provides no return code to the caller and never waits for I/O to complete before returning.

Providing Notification of I/O Completion

The caller is notified of I/O completion (or of an error associated with the request) by the device driver's call to the **iodone** kernel service. A residual count of the number of requested bytes not transferred by the operation is placed in the buffer header **b_resid** field before the I/O is marked complete for the buffer header. If all requested bytes were transferred, this count has a value of 0. Otherwise, it contains the number of bytes that were not transferred.

The device driver indicates an error by setting the **B_ERROR** flag in the associated **buf** header **b_flags** field and placing the error number in the **b_error** field. These fields must have been set before calling the **iodone** kernel service.

The **B_DONE** flag in the buffer header must not be set by the device driver. The **iodone** service sets this flag when called by the device driver and invokes the iodone routine (specified in the buffer header) from the **iodone** interrupt handler. The address of the iodone routine is placed in the buffer header by the caller of the strategy routine, before calling the strategy routine. The device driver calls the **iodone** service for each buffer header received by the strategy routine.

Reordering of I/O Requests

Multiple buffer headers can also be presented to the strategy routine, where the additional buffer headers may be chained to the first by using the **av_forw** pointers. The buffers are not typically back-linked using the **av_back** pointers. While the device driver strategy routine is free to rearrange the buffers on its device queue with respect to the processing for strategy requests, the ordering of the buffer headers provided in a chain to the strategy routine cannot be modified. Therefore, although the device driver might, for optimization purposes, reorder individual strategy requests, the ordering of the transfers within a particular strategy request cannot be changed.

Besides reordering the strategy requests for performance, some strategy routines attempt to coalesce requests into fewer and larger I/O requests. This is possible when the requests can be ordered so that they specify contiguous blocks on the device within the limits of the maximum transfer size of the hardware. See Spanned (Consolidated) Commands in the Execution of SCSI I/O Requests on page 12–5.

Handling Out of Range Block Numbers

The strategy routine also determines if the block number requested in the buffer header is valid for the device. On read operations, a block number at the `end-of-media` is not considered an error, but no data is transferred. For write operations, if the block number is at the `end-of-media`, it is considered an error. In this case, the **B_ERROR** flag in the **buf** structure should be set, and the **b_error** field should be set to contain the ENXIO value.

For both reads and writes, a block number past the `end-of-media` is considered an error. In this case, the **B_ERROR** flag should be set to on, and the **b_error** field should be set to ENXIO. For both reads and writes, if the beginning block number is before the `end-of-media`, and the transfer length causes the `end-of-media` to be reached, then no error is indicated. The **b_resid** count is set to the number of requested bytes that were not transferred because the `end-of-media` was reached. The `end-of-media` is defined as the first block outside the capabilities of the device.

Queuing a Request to the Start I/O Routine

To maintain the state of the device and its I/O requests, the device driver typically allocates a private data structure in system memory associated with the device. Here the device status flags are maintained along with device error information and device queue pointers. Some device drivers maintain more than one queue of buffer headers: one queue for those that are waiting for I/O to start and another for those that currently have I/O in progress.

The buffer headers on these device queues are chained together using the **av_forw** and **av_back** fields in the buffer header. Generally, these fields are used by the caller of the strategy routine to manage the free list of buffer headers. These fields can also be used by the device driver because buffer headers handed to the strategy routine are no longer on a free list.

The Start I/O Routine

Once the strategy routine has queued the buffer headers, it calls the start I/O routine to start processing the I/O requests if the queue had previously been empty. If the queue was not empty, it simply queues the requests without calling the start I/O routine. Queuing the requests is normally done in an interrupt-disabled condition to ensure serialization with the device-handling routines executing in the interrupt environment. This maintains the consistency of the I/O queue.

Once the I/O-handling routines have completed an I/O transfer and performed the corresponding **iodone** processing (typically in the interrupt environment), the device I/O queue is checked to determine if any further requests are queued. If more work is found on the device I/O queue, the start I/O routine is then called from the completion-handling routines.

The start I/O routine is responsible for splitting the I/O transfer requests into multiple I/O transfers, if necessary, and for providing an interface to the hardware. This interface sets up the system and device hardware for the command and data transfers. This can involve preparing for direct memory access (DMA) transfers, performing programmed I/O to the hardware, handling (and possibly retrying) I/O errors, and processing device interrupts.

Understanding Raw I/O Access to Block Device Drivers

While character device drivers can only be accessed by character special files, most block device drivers provide a character as well as a block interface. A dual interface of this kind requires conditions:

- The block device must have both a character and a block special file name so that it can be referenced by either.
- The block driver must have read and write entry points as well as a strategy entry point. The character entry points allow reading and writing of non-cached data.

A block device driver that provides character device entry points is said to provide *raw I/O* support. For example, the first diskette drive is a block device with two special file names. The drive can be accessed as either **/dev/fd0** (block) or **/dev/rfd0** (character). The **r** in the name **/dev/rfd0** stands for raw because character-oriented access to a block device is called *raw I/O*.

Motivation for Providing a Raw I/O Interface to a Block Device

The raw interface to a block device is provided to avoid the buffering usually done for block data transfers. When an I/O request is issued to a block special file, the file system invokes the buffer storage services provided by the kernel. An alternative is to issue the I/O request using the character special file name for the same block device. This circumvents the buffer management services altogether.

Raw interfaces are typically used when a known amount of data is to be transferred. Examples of such cases are when formatting devices or while performing backups (such as backing up a disk to tape). In these cases the amount of data to be read from or written to the device is known ahead of time. The data can be transferred out of the memory buffer allocated by the user process and directly to the device. This transfer can be done without the use of buffers and in blocks as large as the user requests.

Avoiding the buffer storage services used by the file system may result in better performance. This is because the user's buffer size is often more appropriate to the device or operation being used. The raw interface to block device drivers also avoids making an extra data copy when moving the data from user to kernel buffer storage (or vice versa).

Note: Applications should use the raw mode interface to a block device driver carefully. Inconsistencies can arise if device data is accessed in raw mode while a separate version of the data is already present in the kernel's buffer cache. This situation can arise either when a file system is mounted on the device or when access is made through the block special file.

Understanding Raw I/O Support

A mechanism is provided by which block device drivers can provide the ability to transfer data directly between the user's memory and the device. That is, data transfer does not use the block I/O buffer cache and can occur in blocks as large as the caller requests. This mechanism uses a character device special file to provide access to the raw device along with **ddread** and **ddwrite** entry points provided by the block device driver. Instead of sending the read and write requests through the block I/O buffer cache mechanism (as in the case of I/O using the block special file), the requests are processed and sent to the device driver in the same manner as for a character device driver. The device driver **ddread** and **ddwrite** routines, however, are usually much different from the read and write routines typically found in a true character device driver.

Processing a Raw I/O Request

The block device driver's read and write routines typically convert the raw I/O request into a block request that is sent to the device driver's own **ddstrategy** routine. To do this, one or more **buf** headers must be allocated to contain the block requests. These **buf** headers can be created from the kernel heap or allocated from the **buf** header pool used by the block I/O buffer cache.

The device driver routines then process the parameters for the raw I/O data transfer, as provided by the file system in the **uio** structure. Information taken from this structure is transformed into block I/O parameters that are put in one or more **buf** headers. The device driver's own strategy routine is then called to process the I/O request, while the read and write routines typically wait for I/O completion.

The major part of the effort described previously is in converting the raw I/O request into a block request and then making the data buffer in the user address space accessible to the device driver's device handling routines. Because the block I/O is asychronous, the user's buffer must be attached to and accessed using the cross memory services provided by the kernel.

Processing by the uphysio Kernel Service

The kernel provides the **uphysio** kernel service to help convert the request to a block format and provide the cross-memory access. The device driver can indicate the number of **buf** headers that the **uphysio** service is to use. The **uphysio** service then allocates this number of headers, using them to send the requests to the strategy routine. Multiple **buf** headers help maximize the I/O redrive capability of the device's strategy routine. Alternatively, the driver can specify only a single **buf** header to make error handling and recovery simpler.

The device driver can also provide the **uphysio** service with a special parameter-adjusting **mincnt** routine. This routine is called to handle device-dependent restrictions before the **uphysio** service sends the **buf** header to the strategy routine.

Once I/O specified for the current use of the header has finished, the **uphysio** service continues reusing **buf** headers until the entire I/O operation has completed. Completion is achieved when all the data requested in the **uio** structure has been transferred by the strategy routine or when an error is detected. In either case, the **uphysio** routine will not return to the caller until all the I/O transfers it initiated have been completed.

Understanding Character I/O Device Drivers

A device driver in the *character class* supports devices that do not fall into the block I/O model. Character devices such as displays, keyboards, printers, terminals, communications lines, and many pseudo-devices support character-at-a-time I/O. Character device drivers, however cannot support mounted file systems or paging devices. They are generally used by user-mode application programs or device subsystems to access a device.

The device switch table contains the entry point addresses of the interface routines for each character device driver in the system, just as it does for block device drivers. Like block device drivers, character device drivers supply a config routine (the **ddconfig** entry point) for configuration support as well as open and close routines (**ddopen** and **ddclose**) called on each open and on the final close of a device.

If multiplexing is supported, an mpx routine (**ddmpx**) must also be included that is called before the open routine and after the close routine. Character device drivers can also provide an ioctl routine (**ddioctl**) to support special control requests and a revoke routine (**ddrevoke**) if the supported device is considered to be in the Trusted Computing Path. Unlike block device drivers, character device drivers provide read and write routines (**ddread** and **ddwrite**) in the device switch table to process read and write requests directly.

Unsupported Entry Points

A character device driver need not provide all of these routines if they are not required for the device being supported. If a routine is not provided and should not generate an error when called, the corresponding entry point in the device switch table should specify the **nulldev** entry point. If the call should result in an error return, the **nodev** entry point should be specified. These default routines are provided as part of the base kernel. Because both character and block device drivers use the same device switch table, character device drivers should set the block device entry points, **ddstrategy** and **dddump**, to the **nodev** entry point.

Non-multiplexed Support

For traditional, nonmultiplexed character device drivers, the driver's open routine is called for each open of the device, with the device major and minor number and the open mode flags sent as parameters. The close routine is called only when the device is closed for the last time (that is, when the last process for which the device is open closes it). It is therefore not possible for a nonmultiplexed device driver to maintain its own count of its users. The open and close support for multiplexed character device drivers is somewhat different, however.

Multiplexed Support

A multiplexed character device driver provides an mpx routine, whose **ddmpx** entry point is specified in the device switch table. When a device open is processed, the kernel calls the driver's mpx routine before calling the open routine. This mpx routine is called to evaluate any channel name specified with the character special file name. The mpx routine's job is to allocate a channel, associate it with the channel name, and return a channel identifier to the kernel. The kernel then calls the driver's open routine with device major and minor number, control flags, and the supplied channel identifier. If any further device requests (read, write, or ioctl operations, for example) specify this open channel, the kernel provides the corresponding channel identifier to the device driver routine as a parameter.

Unlike a nonmultiplexed character device driver, the multiplexed driver's close routine is called once for each close associated with an explicit open request. For close requests resulting from inherited opens (due to **fork** or **dup** subroutine calls), the driver's close routine is not called. Once the last close for a channel has been processed by the device driver's close routine, the kernel calls the device driver's mpx routine with the channel identifier, requesting that the channel be deallocated.

Read and Write Support

When the read or write routines (**ddread** and **ddwrite**) are called, they are supplied with the device major and minor number, a channel identifier (if multiplexed) and a pointer to a user I/O structure containing the parameters of the read or write request. This **uio** structure contains the following information:

- Number of characters to transfer
- I/O mode flags
- Address and length of one or more data buffers to be used in the I/O
- Address space identifier describing the address space in which the buffer(s) resides.

The address space identifier is provided because these routines can be called by other device drivers. As a result, buffers can be either in system address space, the user's address space, or in a cross-memory address space.

Writing One Character at a Time

The driver's write routine (the **ddwrite** entry point) is responsible for copying characters from the buffer or buffers specified in the **uio** structure to the device. (The number to copy is specified by the **uio_resid** field in the **uio** structure.) For many drivers working with one character at a time, the **uwritec** kernel service can be used for this purpose. This service uses the **uio** structure to retrieve characters from the caller's buffers, which are in the address space designated by the **uio_segflg** field. Successive calls to this service return characters from these buffers until no more characters are available or until an error is detected. The **uwritec** service updates the **uio_resid** field, which is used by the caller of the write routine to determine how many characters were transferred.

Reading One Character at a Time

The driver's read routine (the **ddread** entry point) is responsible for copying **uio_resid** characters from the device to the buffers specified in the **uio** structure. For many drivers working with one character at a time, the **ureadc** kernel service can be used for this purpose. This service uses the **uio** structure to put characters into the caller's buffers, which are in the address space designated by the **uio_segflg** field. Each successive call to this service writes characters into the next available buffer location described by the **uio** structure. This can continue until the **uio_resid** character count is 0 or an error is detected. The **ureadc** service updates the **uio_resid** field, which is used by the caller of the read routine to determine how many characters were transferred.

Moving Large Numbers of Characters at a Time

The **uiomove** kernel service is available for read and write routines that transfer large numbers of characters between the caller's buffer (described by the **uio** structure) and the device (or a device driver buffer). When many characters must be transferred, the use of this service provides much faster transfer of data than the character-at-a-time services. This service either transfers as many characters as its *n* parameter specifies or keeps transferring until the **uio.uio_resid** field becomes 0 (whichever comes first). The direction of the move is specified by the setting of the *rw* parameter on the call to the service.

I/O Control (ddioctl) Support

The I/O control or ioctl routine (the **ddioctl** device driver entry point) is usually provided by a character device driver providing special control functions. The ioctl routine is provided with the device major and minor number, the channel identifier (if multiplexed), an I/O control command parameter, and an argument parameter associated with the command. The meanings of the command and argument parameters are by definition device-specific. However, all device drivers in AIX should support the IOCINFO **ioctl** operation, which provides general device information. In addition, **tty** device drivers generally support a base set of I/O control commands defined in the general terminal **termio** interface.

Select and Poll Support

Character device drivers can also support multiple I/O event notification by providing a select routine (the **ddselect** device driver entry point) in the device switch table. This routine is invoked in response to a **select** or **poll** subroutine call with the device major and minor number, channel identifier (if multiplexed), a requested events parameter, and a returned events parameter. Flags in the requested events parameter indicate which event is being requested along with a synchronous request indication. The most commonly supported events are data available for reading (POLLIN), device available for writing (POLLOUT), and exceptional condition outstanding (POLLPRI).

The select routine should check the current state of the device and set the corresponding flag or flags in the returned events parameter. If at least one requested event is indicated as true in the returned events parameter, or if the synchronous request flag is set in the requested events parameter, the select routine should simply return from the call.

If none of the requested events are true and the synchronous request flag is not set, the select routine should remember which events have been requested for this device (by setting state flags in a private data area) and return to the caller. Other device driver routines, typically interrupt handlers, should check the requested-event state flags and notify the system if one or more of the events have become true for the device.

Notification of the event is achieved by calling the **selnotify** kernel service. This service takes as input the device major and minor number, channel number (if multiplexed, or 0 if not), and a returned events parameter indicating which events have become true for the specified device. Unlike previous UNIX support for this capability, requesting-process collisions and process identifiers do not have to be dealt with by the device driver. The **selnotify** kernel service wakes up all processes still waiting on one or more of the events now true for the device specified. After calling the **selnotify** kernel service, the device driver should reset the requested state flags for the events that have become true.

Note: The synchronous request flag and the requested-event state flags are used and maintained by the device driver for performance reasons. These fields are used to prevent unnecessary calls to the **selnotify** kernel service, such as when events on a device are no longer being waited for. Actually, the **selnotify** kernel service knows not to perform notification in these cases and could be called even when the original request was synchronous, or for devices and events that were not requested.

Although calling the **selnotify** routine in all these cases might make device driver programming simpler, it could have adverse effects on device and system performance. This is because the **selnotify** routine must search a hash chain for events and devices not present each time it is called. It is recommended that the programmer of this routine ensure optimal device and system performance by using the synchronous request flag and maintaining requested event state information.

Device drivers providing a select routine can also use other device drivers, perhaps as device handlers. The kernel provides a *cascading* select kernel service called **fp_select** that can be used to pass select requests from one device driver to another.

Trusted Computing Path Support

Device drivers supporting terminal (display or keyboard) I/O devices on the AIX Base Operating System should provide a revoke routine (the **ddrevoke** entry point) in the device switch table. When called by the kernel, this routine should terminate any processes sleeping in the device driver (they are typically waiting on I/O) by issuing the **signal** subroutine call with the **SIGKILL** signal. This should be done for each process put to sleep by the device driver that is waiting on the designated device. The revoke routine is used by the security services in the AIX Base Operating System.

Physical Device Support

Character device drivers supporting physical devices have device I/O routines and device interrupt handlers that provide an interface to the hardware. These routines are used by the open, close, read, write, and I/O control routines. The device-handling routines are generally in the bottom half of the device driver, which can be executed in both the process or interrupt handler environment.

Most device drivers use buffering mechanisms and queues between the device head routines in the top half of the device driver and the device handler routines in the bottom half. For relatively low data-rate devices, a character list (**clist**) buffering mechanism is provided by the kernel's **clist services**, which can be used by device driver top and bottom half routines. This kernel-provided set of character buffers is shared among all character-oriented devices that use these **clist** services. Because a limited amount of character buffer space is provided, device drivers should maintain a maximum character queue depth. This avoids excessive use of the available space by one device.

For high data-rate devices, the device driver programmer can choose to implement a private buffering scheme. This can be achieved by allocating memory from the kernel or pinned kernel heap using the **xmalloc** kernel service. When this memory is no longer being used, it must be returned using the **xmfree** service.

Alternately, some character device drivers of this type *borrow* buffers from the block I/O buffer cache pool provided by the kernel. The **geteblk** kernel service can be used for this purpose, but the resources of this pool are also limited. Overuse of these buffers can impair the performance of block I/O devices, because fewer buffers are available for block I/O device caching. When these borrowed buffers are no longer in use, they should be returned to the buffer pool by using the **brelse** kernel service.

Understanding Off-Level Processing

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 degraded seriously.) 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 which depicts interrupt priority versus interrupt service times:

Priority	**Service Time (machine cycles)**
INTCLASS0	200 cycles
INTCLASS1	400 cycles
INTCLASS2	600 cycles
INTCLASS3	800 cycles
INTOFFL0	1500 cycles
INTOFFL1	2500 cycles
INTOFFL2	5000 cycles
INTOFFL3	5000 cycles.

Off-Level Interrupts

The INTOFFL*n* interrupt priorities are for off-level interrupt processing. Typically, they are used when the interrupt service time for an operation exceeds the time allowed at that interrupt priority. The **i_sched** kernel service is used to schedule off-level processing. The operation is then set up to be performed at an off-level interrupt priority. This allows other device interrupts to preempt the operation of the off-level handler at a small cost of additional system overhead.

Operations that do not meet the off-level service time requirements must be scheduled to be performed under a kernel process in order to maintain adequate system real-time performance.

Device driver routines providing the device handler role often include an off-level processing routine. The kernel calls the off-level routine to perform device-specific processing after the following events have taken place:

- The interrupt handler has completed its processing.
- The interrupt has been reset.

The processing associated with a device interrupt can be time-consuming. The off-level routine allows a device to perform this processing at a less favored priority. This in turn enables interrupt handlers to run as fast as possible by avoiding interrupt-processing delays and device overrun conditions.

This routine must be part of the bottom half of the device driver when present.

Understanding Pseudo-Device Drivers

The AIX operating system supports and uses the concept of pseudo-devices. Device drivers for pseudo-devices are used to access low-level system facilities that are not necessarily true I/O devices. These system facilities can be purely software functions for which there is no associated physical device.

Pseudo-device drivers are accessed by special file path name, just as regular device drivers are. As a result, access to particular system facilities can be controlled by the access permission mode of the special file. Thus pseudo-device drivers are provided because the I/O subsystem model provides a convenient way to control access to these software functions.

For example, the **mem** and **kmem** pseudo-device drivers allow user applications access to memory that is not ordinarily accessible in their protection domain or address space. (This assumes that they do, however, have the required privilege).

The AIX Base Operating System provides the following special-purpose drivers among others:

tty	Allows a program to access its controlling terminal.
null	Discards output written to it and indicates an end-of-file condition when read.
bus	Permits direct access to the I/O bus for memory-mapped I/O.
mem	Provides access to system memory.
kmem	Provides access to kernel memory.
trace	Records data when tracing programs.
console	Provides access to the system console.
dump	Provides identification of dump devices and control of system dumps.

I/O Exception Handling Overview

The AIX Base Operating System handles errors caused by programmed I/O as synchronous exceptions. This means that the error notification immediately follows the instruction with which the error was associated.

Asynchronous errors, such as errors during DMA operations, do not generate exceptions. Instead, these errors are detected when the processor performs a status check of the operation. (For example, the **d_complete** kernel service performs such checks.)

Hardware error handling, logging, and recovery are very hardwareplatform-specific in nature. However, the AIX kernel provides a general mechanism and structure that can be used for error recovery on many hardware platforms.

Device Handler Error Recovery

Device handlers performing programmed I/O in an AIX environment are expected to set up exception handlers to catch and log errors due to programmed I/O. Failure to catch an exception of this nature generally causes the default exception handler to be invoked. This default exception handler normally logs the error and crashes the system (if the error was generated from kernel mode). A device handler must therefore register an exception handler (by using the **pio_assist** or **setjmpx** kernel services) to catch I/O exceptions even if the handler cannot recover the state of the device.

Device handlers supporting devices that never generate bus errors must also handle bus errors that can occur while performing programmed I/O with their device. This is because bus errors detected by other devices on the Micro Channel can generate an exception due to detected errors on the bus, even though the device detecting the error is not itself the target of the I/O operation.

The kernel's first-level exception handler obtains error information and invokes the most recently registered exception handler. If recovery of the device is not possible, the device handler should return an error indication to the routine requesting the failed I/O operation.

Recoverable Hardware I/O Errors

The following errors are designated as generally recoverable on the RISC System/6000 platform for most adapters:

- I/O Adapter activated Channel Check Line
- Parity error occurred on bus controller resources
- Data Parity Error was detected on read from adapter
- No Response from adapter was detected on read/write.

Non-recoverable Hardware I/O Errors

The following errors may occur on the RISC System/6000 but are not typically recoverable by a device handler. Errors of this class usually indicate programming errors:

- I/O Register access attempt without proper authority
- I/O Bus memory access attempt without proper authority
- I/O Bus memory access caused Page Fault.

Typical device handler action when encountering an exception of this class is to log the failure and provide an error code to the user of the device at the time of the error.

For device handlers not using the **pio_assist** kernel service, exception conditions due to I/O bus exceptions cause a return from the **setjmpx** kernel service with a return code of **EXCEPT_IO**. The **getexcept** kernel service returns a structure containing platform-specific error information. This information is described in the definition of the **pio_except** structure found in the **<sys/except.h>** header file for the appropriate platform.

For the RISC System/6000, this error information allows the user to determine the following:

- Which of the previously mentioned errors above occurred
- Whether the operation was a load or store
- The bus unit ID used
- The effective address used on the access.

Interfacing to the Hardware

The following discussion topics are provided as guidance for accessing and controlling I/O devices from a device driver:

- Processing Interrupts
- Understanding Direct Memory Access
- I/O Exception Handling Overview.

Processing Interrupts

An interrupt level is the means by which a device notifies the system of the occurrence of an event. How interrupt levels are assigned to an adapter depends on the type of bus to which the adapter interfaces.

Some bus implementations allow interrupt levels to be assigned at system configuration time. System configuration software determines which adapters are present and assigns an interrupt level to the device adapter using special bus commands. System configuration then sets the device configuration and initialization data to reflect this assignment.

However, some buses do not support programmable assignment of interrupt levels. The assignment of these interrupt levels is usually hardwired or selected by a jumper on the adapter. In the latter case, system configuration executes an adapter-specific command that determines how the adapter is configured. The device's configuration and initialization data is then set to reflect the adapter's configuration.

The RISC System/6000 supports I/O adapters attached to the Micro Channel Bus. This bus and associated adapters support POS (a Programmable Option select capability). The POS capability allows the adapters to be configured into the system using software instead of hardware switches and jumpers.

Each time the System/6000 is booted, the Micro Channel Bus configuration method scans the bus and creates a list of all adapter cards plugged into the slots. For each adapter plugged into a slot, the method uses the adapter ID (sensed from the POS registers) to look up the adapter's assignable resources in the devices database.

If the adapter uses one or more interrupt request lines, database adapter attributes describe all possible interrupt level assignments to which the adapter can be programmed. A default or preferred interrupt level is also given. An attribute associated with each interrupt request line indicates which interrupt priority an interrupt line should be assigned. Interrupt levels can be shared by more than one adapter if and only if they all request the same interrupt priority. The system does not support different priorities on the same interrupt level.

The bus configuration method selects a interrupt level assignment for each adapter using interrupts in the system so that no interrupt level is assigned two different priorities. These assigned interrupt levels are then written into the Customized Devices database object for each adapter in a slot. Interrupt priority assignments are assumed to be fixed and are never modified by the configuration program.

When the adapter's specific configuration method is called later in the configuration process, it reads the assigned interrupt level and associated priority from the database for the specific adapter being configured. The adapter's method then puts this information in a device-dependent structure used to initialize the device driver supporting the adapter.

When the device driver is initialized for the adapter in the specified slot, the information in the device-dependent structure is written to the adapter's POS registers. This action properly configures the adapter.

Kernel Services for Managing Interrupts

The AIX kernel provides the following kernel services for managing interrupts.

i_init	Allocates an interrupt level.
i_unmask	Enables an interrupt level.
i_mask	Disables an interrupt level.
i_clear	Frees an interrupt level.
i_enable	Enables interrupt priorities.
i_disable	Disables interrupt priorities.
i_sched	Schedules off-level processing.

The **i_enable** and **i_disable** services should be used to serialize the execution of device driver code with its interrupt handler. The **<sys/intr.h>** header file defines the valid interrupt priorities. It also indicates the interrupt priorities that various kernel services use to serialize their execution.

The **i_sched** service can be used to schedule some of a device driver's interrupt processing at a less favored interrupt priority.

Both bus and off-level interrupt handlers have guidelines for maximum pathlengths. Understanding Interrupts on page 6–9 provides more information about interrupt priorities and maximum path length. Understanding Off-Level Processing provides the guidelines for off-level handler path length.

Early Power-Off Warning

Some machines detect that power is about to be lost and generate an early power-off warning (EPOW). Some device drivers may need an early power-off warning to recover gracefully from loss of power.

For example, the AIX file system on the RISC System/6000 requires that no sector be damaged when power is lost. To avoid damage, devices containing file system data must be stopped at a sector boundary when power is about to be lost.

A device driver can request that it be notified when an EPOW occurs. To make such a request, the driver must call the **i_init** kernel service to define an interrupt handler for interrupt priority **INTEPOW**. The kernel calls all interrupt handlers thus defined at **INTEPOW** priority when an EPOW occurs.

A device handler should register an EPOW handler if it is critical that data-write operations be halted on specific boundaries for data integrity and recovery. However, the path length and time to halt a device must be extremely short because the amount of time between the early power-off warning and actual power loss is usually very short (a few milliseconds). (This timing is hardwaredependent.) Only critical data devices such as disks should need to register an EPOW handler.

The **INIT_EPOW** macro in the **<sys/intr.h>** header file can be used to initialize the *handler* parameter passed to the **i_init** service for registering EPOW handlers.

The invocation of a registered EPOW interrupt handler is different from that for other interrupt handlers registered by the **i_init** service. There are three conditions under which registered EPOW handlers are called:

EPOW_SUSPEND	Invocation is due to an Early Power Off Warning (EPOW) without battery backup, or when the battery backup is exhausted. Critical device operation should be suspended. Interrupt handlers are called at INTEPOW priority. The **EPOW_SUSPEND** flag is set in the **flags** field of the **intr** structure pointed to by the *handler* parameter when the interrupt handler is called.
EPOW_BATTERY	Invocation is due to an Early Power Off Warning (EPOW) resulting in a switch-over to backup battery power. Devices not configured for battery backup operation should be suspended. EPOW interrupt handlers are called at INTEPOW priority. When calling the interrupt handler, the kernel sets the EPOW_BATTERY flag in the **flags** field of the **intr** structure pointed to by the *handler* parameter .
EPOW_RESUME	Invocation is due to a restoration of power. Any operations suspended due to previous EPOW_SUSPEND or EPOW_BATTERY conditions should be resumed. This normally occurs when either of the following is true: • The early power off warning was a false one caused by a power fluctuation that did not actually cause loss of power. • The system was running on battery backup and primary power is restored. Interrupt handlers are called at the INTTIMER priority for this function.

Device handlers are responsible for ensuring the proper serialization of operation when handling EPOW interrupts, normal device interrupts, and process level operations. Following are possible complications.

An early power-off warning can prove to be a false alarm. If this happens, the EPOW interrupt handlers are called to suspend device operation (at a high priority) and later called at a lower priority to resume device operation. If power is actually lost, the EPOW_RESUME operation does not occur.

A second early power-off warning can be detected while trying to resume from an earlier one. When this situation arises, an EPOW interrupt handler can be reinvoked during the course of an EPOW_RESUME call by the higher priority EPOW_SUSPEND or EPOW_BATTERY calls. In this case, EPOW interrupt handlers may find that both the **EPOW_SUSPEND** (or **EPOW_BATTERY**) and **EPOW_RESUME** flags are set in the **flags** field within the **intr** structure. If this situation is detected, the suspend operation should occur and the resume request should be ignored.

The EPOW interrupt handlers should ensure that no timing window can occur in which device operation is restarted after an EPOW_SUSPEND condition and before an EPOW_RESUME condition. This must not happen even if a suspend operation interrupts a resume. To prevent this situation, the EPOW handler should check the **EPOW_SUSPEND** and **EPOW_RESUME** flags in the **intr** structure, and then determine if the device is already in a suspended state. (A device driver flag should be maintained for this purpose.) If this is a suspend call and the device is already in the suspended state, no operation should be performed. If this is a resume request and the device is suspended, the `device-suspended` flag should be reset and the device started.

The EPOW interrupt handlers should ensure that no timing window can occur in which device operation is restarted after an EPOW_SUSPEND condition and before an EPOW_RESUME condition has. This situation can arise when the suspend operation interrupts the resume. To prevent this situation, the EPOW handler should check the **EPOW_SUSPEND** and **EPOW_RESUME** flags in the **intr** structure, and then determine if the device is already in a suspended state. (A device driver flag should be maintained for this purpose.) If this is a suspend call and the device is already in the suspended state, no operation should be performed. If this is a resume request and the device is suspended, the `device-suspended` flag should be reset and the device started.

Note: The check for the **EPOW_SUSPEND** or **EPOW_BATTERY** flag and the checking and clearing of the `device-suspended` flag should be made an atomic operation by performing them at INTEPOW priority. Doing so ensures that an intervening EPOW_SUSPEND or EPOW_BATTERY operation does not result in the device being resumed during an EPOW_RESUME condition.

Such atomic operations also require that the device hardware support a state in which a pending operation is not started. For a SCSI device, a SCSI Reset and resulting Unit Attention provide this state. Other devices may require SUSPEND and RESUME hardware commands.

Direct Memory Access (DMA)

The Micro Channel supports two types of DMA adapters. These are DMA slaves and DMA masters. A DMA slave adapter is the simpler form of adapter. It requires extensive system support to generate addresses and control the transfer length. The system hardware limits a DMA slave adapter to performing only one sequential transfer at any one time.

A DMA master generates its own bus address and controls its own transfer length. A DMA master adapter is therefore only limited by its own hardware in the number and type of transfers that it can perform. For example, a DMA master disk adapter can support one or more concurrent DMA transfers for each disk connected to it. A DMA master LAN adapter can support having the header at one location in system memory and the data at another location.

Block DMA Transfers

A block DMA transfer consists of transferring data between sequential locations on the adapter and sequential locations in memory. All DMA slaves are essentially limited to this type of transfer.

A DMA slave can have only one contiguous block transfer in progress at any one time. The maximum size of this transfer is machine-dependent and is defined in the **<sys/dma.h>** header file.

A DMA master can have one or more block transfers in progress at any one time. Each transfer must be assigned part of that DMA master's fixed-size window into system memory. This window is assigned to the adapter during system configuration.

The device driver can manage the use of this window in any way that is appropriate. Typically, each active request is assigned part of this window through which to perform its data transfer. Requests waiting to be processed are not yet assigned to a part of the window.

Device drivers may also use part of their window to provide system memory access to their adapter for control and status information.

A device driver must call either the **d_slave** service to set up a DMA slave transfer or the **d_master** service to set up a DMA master transfer. The device driver should then set up the device to perform the DMA transfer. The device transfers data when it is available and interrupts the processor upon completion of the DMA transfer. The device driver then calls the **d_complete** service to clean up after the DMA transfer. These steps are typically repeated each time a DMA transfer is to occur.

DMA Management Kernel Services provides more information on using the DMA kernel services.

DMA Processing

Direct memory access (DMA) allows a device to access memory without going through the processor. Using DMA consists of the following steps:

1. Allocating a DMA channel.
2. Initializing the DMA channel.
3. Enabling the DMA channel.
4. Performing one or more DMA transfers.
5. Disabling the DMA channel.
6. Freeing the DMA channel.

The DMA transfer itself, in Step 4 previously, consists of the following steps:

1. Arbitrating for the bus.
2. Generating an address.
3. Performing the data transfer.

The AIX kernel provides a set of services that assist in performing DMA operations. DMA Management Kernel Services provides more information on using these services.

DMA Channels and How They Are Assigned

A DMA channel is the means by which DMA transfers for different adapters are distinguished from each other. A DMA channel is a resource that cannot be shared simultaneously by two adapters.

How DMA channels are assigned to an adapter depends on the type of bus to which the adapter interfaces. The Micro Channel allows for assignment of DMA channels at system configuration time. System configuration software determines which adapters are present and assigns a DMA channel to the device adapter. System configuration then sets the device configuration and initialization data to reflect this assignment.

However, some buses do not support programmable assigment of the DMA channel. DMA channel numbers are hardwired or selected by a jumper on the adapter. In this case system configuration executes an adapter-specific command that determines how the adapter is configured. The device's configuration and initialization data is then set to reflect the adapter's configuration.

The RISC System/6000 supports I/O adapters attached to the Micro Channel Bus. This bus and associated adapters support POS (a Programmable Option Select capability). The POS capability allows the adapters to be configured into the system using software instead of hardware switches and jumpers.

Each time the System/6000 is booted, the Micro Channel Bus configuration method scans the bus and creates a list of all adapter cards plugged into the slots. For each adapter plugged into a slot, the method uses the adapter ID (sensed from the POS registers) to look up the adapter's assignable resources in the devices database.

If the adapter uses the DMA channel, the database describes all possible DMA channels to which the adapter can be programmed and a default or preferred choice. The bus configuration method then selects a unique DMA channel for each adapter requiring DMA in the system. The assigned DMA channel numbers are written into the Customized Devices database object for each adapter in a slot.

When the adapter's specific configuration method is called later in the configuration process, it reads the assigned DMA channel or channels from the database for the specific adapter being configured. The adapter's configuration method then puts these channels in a device-dependent structure used to initialize the device driver supporting the adapter.

When the device-driver for the adapter in the specified slot is initialized, the information in the device-dependent structure is written to the adapter's POS registers. This action properly configures the adapter.

Kernel Services for Performing DMA Transfers

DMA Management Kernel Services provides more information on using these services.

Installing and Configuring Device Drivers

The following topics are available for guidance in installing and configuring device drivers:

- Program Installation and Update Compatibility Overview
- The Device Configuration Subsystem: Programming Introduction.

Files

<sys/dma.h>

<sys/intr.h>

Related Information

The **open** subroutine, **close** subroutine, **read** subroutine, **write** subroutine, **lseek** subroutine, **ioctl** subroutine, **select** subroutine, **poll** subroutine, **signal** subroutine.
The **ddconfig** device driver entry point, **ddopen** device driver entry point, **ddclose** device driver entry point, **ddread** device driver entry point, **ddwrite** device driver entry point, **ddioctl** device driver entry point, **ddrevoke** device driver entry point, **ddmpx** device driver entry point, **ddselect** device driver entry point, **ddstrategy** device driver entry point, **dddump** device driver entry point..
The **uiomove** kernel service, **ureadc** kernel service, **uwritec** kernel service, **xmalloc** kernel service, **xmfree** kernel service, **geteblk** kernel service, **brelse** kernel service, **selnotify** kernel service, **iodone** kernel service, **i_sched** kernel service, **uphysio** kernel service, **i_init** kernel service, **i_unmask** kernel service, **i_mask** kernel service, **i_clear** kernel service, **i_enable** kernel service, **i_disable** kernel service.
The **mincnt** routine.
The **clist** structure, **uio** structure, and **buf** structure.

The **d_init** kernel service, **d_unmask** kernel service, **d_mask** kernel service, **d_clear** kernel service, **d_master** kernel service, **d_slave** kernel service, **d_complete** kernel service.
Device Driver Roles on page 2–2, Device Driver Structure on page 2–3, Understanding I/O Access Through Special Files on page 2–4, Device Driver Classes on page 2–1, Understanding the Device Switch Table on page 2–7, Understanding Major and Minor Numbers on page 2–7.
Block I/O Buffer Cache Kernel Services on page 6–8.
Kernel Environment Programming on page 1–1.
Device Driver System Dump Support in *Files Reference*
Device Driver Concepts Overview on page 2–1.
Special Files Overview in *Files Reference.*
Cross Memory Kernel Services on page 6–15.
DMA Management Kernel Services in The I/O Kernel Services.
Configuration Subsystem on page 7–1.
Adapter-Specific Considerations for the Predefined Attribute (PdAt) Object Class in *Files Reference.*
ODM Device Configuration Object Classes in *Files Reference.*

Writing System Calls

The topics below are provided as guidance in writing system calls. The first two articles introduce system calls within the context of kernel architecture. The remaining subjects detail aspects of the execution environment relevant to programming system calls.

The following introductory topics explain the relationship between system calls and user functions in the kernel, as well as details of the system call handler and how system calls are executed:

- Extending the Kernel with System Calls
- Understanding System Call Execution.

The following topics explain aspects of the kernel programming environment that the programmer should consider when writing new system calls:

- Accessing Kernel Data While in a System Call
- Preempting a System Call
- Handling Signals While in a System Call
- Handling Exceptions While in a System Call
- Understanding Nested System Calls and Kernel-Mode Use of System Calls
- Page Faulting within System Calls
- Returning Error Information from System Calls
- System Calls Available to Kernel Extensions.

Extending the Kernel with System Calls

Adding system calls is one of several ways to extend the functions provided by the AIX Operating System kernel. System calls provide user-mode access to special kernel functions. In the AIX operating system, a system call is nothing more than a call that crosses a protection domain.

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

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** may have functions that perform some user-mode processing and then internally invoke a system call. In other cases, the system call can be directly exported by the library without a user-mode layer.

In this way, operating system functions available to application programs may 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.

Programming in the Kernel Environment provides more information on how to use system calls in the kernel environment.

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 executes with kernel-mode privilege in the kernel protection domain.
- A system call's code and data are located in global kernel memory.

- System call routines can create and use kernel processes to perform asynchronous processing. Using Kernel Processes gives more information on creating and using kernel processes.
- System calls are not interruptible by signals.
- System calls cannot use shared libraries or any symbols not found in the kernel protection domain.

Understanding System Call Execution

The system call handler gains control when a user program executes a call to a system call. The system call handler changes the protection domain from the caller's protection domain, *user*, to the system call's protection domain, *kernel*, and switches to a protected stack.

The system call handler then calls the function supporting the system call. The loader maintains a table of the currently defined system calls for this purpose.

The system call executes within the calling process, but with more privilege than the calling process. This is because the protection domain has changed from *user* to *kernel*.

The system call function returns to the system call handler when it has performed its operation. The system call handler then restores the state of the process and returns to the user program.

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

The User Protection Domain

Programs that execute in the *user protection domain* include those executing within user processes and those within real-time processes. This protection domain implies that code executes in user execution mode and has:

- Read/write access to user data in the process private region
- Read access to the user text and shared text regions
- Access to shared data regions using the shared memory functions.

Programs executing in the user protection domain do not have access to the kernel or kernel data segments except indirectly through the use of system calls. A program in this protection domain can only affect its own execution environment and executes in the processor unprivileged state.

The Kernel Protection Domain

Programs that execute in the *kernel protection domain* include interrupt handlers, kernel processes, the base kernel, and kernel extensions (device drivers, system calls, and file systems). This protection domain implies that code executes in kernel execution mode and has:

- Read/write access to the global kernel address space
- Read/write access to the kernel data in the process private region when executing within a process.

User data within the process address space must be accessed using kernel services. Programs executing in this protection domain can affect the execution environments of all programs because they:

- Can access global system data
- Can use kernel services
- Are exempt from all security restraints
- Execute in the processor privileged state.

All kernel extensions execute in the kernel protection domain as described above. The use of a system call by a user-mode process allows a kernel function to be called from user mode. Access to functions that directly or indirectly invoke system calls is typically provided by programming libraries providing access to operating system functions.

Actions of the System Call Handler

When a call is made in user mode that invokes a system call, the system call handler is invoked. This system call handler switches the protection domain from user to kernel and performs the following steps:

1. Sets privileged access to the process private address region.
2. Sets privileged access to the kernel address regions.
3. Sets the **u_uerror** field in the **u-block** (the user structure) to 0 (zero).
4. Switches to the kernel stack.
5. Invokes the specified kernel function (the target of the system call).

On return from the specified kernel function, the system call handler performs the following steps before returning to the caller:

1. Switches back to the user's stack.
2. Updates the **errno** global variable if the **u_error** field is not equal to 0 (zero).
3. Clears the privileged access to the kernel address regions.
4. Clears the privileged access to the process private region.
5. Performs signal processing if a signal is pending.

The system call (and associated kernel function) executes within the context of the calling process, but with more privilege than the user-mode caller. This is because the system call handler has changed the protection domain from user state to kernel state. When the kernel function that was the target of the system call has performed the requested operation (or encountered an error), it returns to the system call handler. When this happens, the system call handler restores the state and protection domain back to user mode and returns control to the user program.

Accessing Kernel Data While in a System Call

A system call can access data that the caller cannot because the system call is executing in a more privileged protection domain. This applies to all kernel data, of which there are three general categories:

- The **user block** data structure

  System calls should use the available kernel services and system calls to access or modify data traditionally found in the **u area** (user structure). For example, the system call handler uses the **u.u_error** system call error field to set the **errno** global 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 may be obtained by using the **getpid** 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 executes on a protected stack that is located near the **user block** data structure at the top of the process private segment. This stack allows the system call handler to safely execute a system call even when the caller does not have a valid stack pointer initialized. It also allows system calls to access privileged information with automatic variables without exposing the information to the caller.

Warning: Great care must be taken in writing system calls that modify fields in kernel or **user block** data structures. Incorrect modifying of fields could cause unpredictable results or system crashes while executing in the kernel protection domain.

Passing Parameters to System Calls

The fact that a system call does not execute on the same stack as the caller imposes one limitation. System calls are limited in the number of parameters that they can use.

The AIX operating system linkage convention passes some parameters in registers and the rest on the stack. The system call handler ensures that the first 8 words of the parameter list are accessible to the system call. All other parameters are not accessible.

Also, care should be taken when defining the interface to a system call. For some languages, various types of parameters may take more than one word in the parameter list. The writer of a system call must be familiar with the way parameters are passed by his or her compiler and conform to this 8-word limit.

Preempting a System Call

The AIX kernel allows a process to be preempted by a more favored process even when executing a system call. This is not typical of most UNIX systems. The AIX kernel makes this change to enhance support for real-time processes and large multiuser systems.

System calls should use the **lockl** and **unlockl** kernel services to serialize access to any global data that they access. Remember that all of the system call's static data is located in global memory and therefore must be accessed serially.

The **lockl** kernel service ensures that the owner of a lock executes with the most favored priority of any of the waiters of that lock. It does this by assigning to the lock owner the process priority of the most favored waiter for the lock. This mechanism is similar to the standard UNIX sleep priority. However, the process priority must be assigned when the resource is allocated since the system call can be inactivated by preemption, as well as by calling sleep. The **unlockl** service restores the process priority.

Note that a process can be preempted even when it owns a lock. The lock only ensures that another process that tries to lock the resource will have to wait until the owner of the resource unlocks it. A system call must never return with a lock locked. By convention, a locking hierarchy is followed to prevent deadlocks. Understanding Locking provides more information on locking.

Handling Signals While in a System Call

Signals may be generated asynchronously or synchronously with respect to the process that will receive the signal. An asynchronously generated signal is one that results from some action external to a process. It is not directly related to the current instruction stream of that process. Generally these are generated by other processes for interprocess communication or by device drivers.

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

Delivery of Signals to a System Call

The kernel delays the delivery of all signals, including **SIGKILL**, when executing a system call, device driver, or other kernel extension. The signal takes effect upon leaving the kernel and returning from the system call. This happens when execution returns to the user protection domain, just before executing the first instruction at the caller's return address. Signal delivery for kernel processes is described in Using Kernel Processes.

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 **lockl**, **e_sleep**, **e_sleepl**, and **e_wait** all support terminating a wait by a signal. These services provide three options:

- The `short-wait-option` of not terminating the wait due to a signal.
- Terminating the wait by return from the kernel service with a return code of `interrupted-by-signal`.
- Executing a **longjmpx** kernel service call to resume at a previously saved context in the event of a signal.

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, AIX kernels automatically saved context on entry to the system call handler. As a result, any long (interruptable) 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 AIX 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 that will set the system call's return code to a –1 and the **u.u_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. It 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 interrupts detected by the processor as a result of the current instruction stream. They therefore take effect synchronously with respect to the current process.

The default exception handling normally generates a signal if the process is in a state where signals are delivered without delay. If delivery of a signal may be delayed, however, default exception handling causes a dump.

Alternative Exception Handling Using the setjmpx Kernel Service

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

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

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

In that case, 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 Nested System Calls and Kernel-Mode Use of System Calls

The AIX Operating System supports nested system calls with some restrictions. System calls (and any other kernel-mode routines executing 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 call system calls that have one or more parameters passed by reference. Doing so may 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 fail 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 may not call the **fork** or **exec** system calls, among others. A list of the base AIX system calls available to system calls or other routines in kernel mode is provided in the List of System Calls Available in the Kernel.

Page Faulting within System Calls

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

- 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 execute while the system call is waiting for the paging I/O to complete.

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

Returning Error Information from System Calls

System calls return error information slightly differently than is the convention for kernel services that are not system calls. System calls typically provide a return code of 0 if no error has occurred, or –1 if an error has occurred. In the latter case, the error value is placed in the **u.u_error** field of the **u area** (user structure). In some cases, when data is returned by the return code, a data value of –1 indicates error. Or alternatively, a value of NULL can indicate error, depending on the interface and function definition of the system call.

In any case, when an error condition is to be returned, the **u.u_error** field should be updated by the system call pror to returning from the system call function. The **u_error** field can be accessed by using the **getuerror** and **setuerror** kernel services.

Before actually calling the system call function, the system call handler sets the **u.u_error** field to 0. Upon return from the system call function, the system call handler copies the value found in **u.u_error** into the **errno** global variable if **u.u_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 may choose to return the error value provided by the nested system call function or may replace this value with a new one by using the **setuerror** kernel service.

System Calls Available to Kernel Extensions

System calls are available either to all kernel extensions or to kernel processes only. System calls are never available to interrupt handlers. The following system calls are available to all kernel extensions:

- **getgidx**
- **gethostid**
- **getpgrp**
- **getppid**
- **getpri**
- **getpriority**
- **getuidx**
- **semget**
- **seteuid**
- **setgid**
- **setgidx**
- **sethostid**
- **setpgid**
- **setpgrp**
- **setpri**
- **setpriority**
- **setreuid**
- **setsid**
- **setuid**
- **setuidx**
- **ulimit**
- **umask.**

The following system calls are available to kernel processes only:

- **disclaim**
- **getdomainname**
- **getgroups**
- **gethostname**
- **getpeername**
- **getrlimit**
- **getrusage**
- **getsockname**
- **getsockopt**
- **gettimer**
- **resabs**
- **resinc**
- **restimer**
- **semctl**
- **semop**
- **setdomainname**
- **setgroups**
- **sethostname**
- **setrlimit**
- **settimer**
- **shmat**
- **shmctl**
- **shmdt**
- **shmget**
- **sigaction**
- **sigprocmask**
- **sigstack**
- **sigsuspend**
- **sysconfig**
- **times**
- **uname**
- **unamex**
- **usrinfo**
- **utimes**.

Related Information

The **getuerror** kernel service, **setuerror** kernel service, **initp** kernel service, **lockl** kernel service, **e_sleep** kernel service, **e_sleepl** kernel service, **e_wait** kernel service, **setjmpx** kernel service, **longjmpx** kernel service, and **unlockl** kernel service.

The **fork** subroutine.

Kernel Environment Programming on page 1–1.

Accessing User Mode Data While in Kernel Mode on page 1–10.

Using Kernel Processes on page 1–7, Using Libraries on page 1–5.

Writing a Device Driver on page 2–1.

Understanding Locking on page 1–11, Understanding Interrupts on page 6–9.

Writing a Virtual File System

The following information is available in understanding virtual file systems:

- Virtual File System Kernel Extensions
- Logical File System Overview
- Virtual File System Overview
- Virtual Nodes (Vnodes)
- Generic Inodes (Gnodes)
- Understanding The Virtual File System Interface.

Virtual File System Kernel Extensions

There are two essential components in the file system:

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.

The virtual file system interface in usually referred to as the *vnode* interface. The vnode structure is the key element in communication between the virtual file system and the layers that call it.

Both the virtual and logical file system exist across all AIX family platforms.

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 AIX kernel with a consistent view of what may 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.

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 UNIX file-system access semantics. This does not mean that only UNIX 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 the following 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 vnode (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 the **fp_ioctl** and **fp_select** routines, that begin with the prefix **fp_**.

- **vnodes**

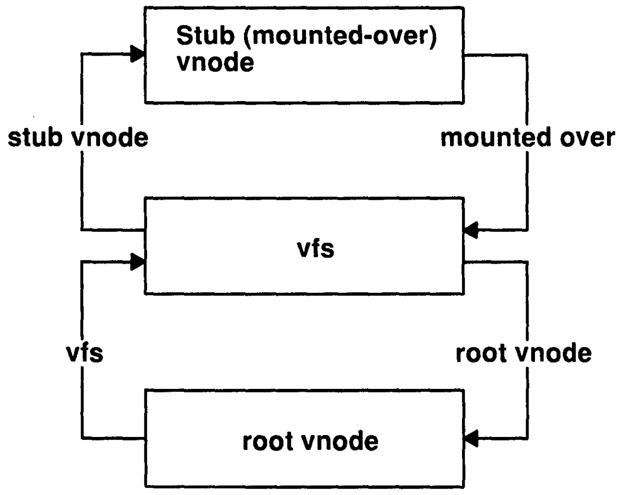
Provide system calls with a mechanism for local name resolution. Local name resolution allows the logical file system to access multiple file system implementations through a uniform name space.

Virtual File System Overview

The AIX 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.

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 vnode (virtual node) and the root of the virtual file system subtree. A mounted-over, or stub, vnode points to a virtual file system, and the mounted VFS points to the vnode it is mounted over.



Virtual File System Mount Point

Virtual Nodes (Vnodes)

A *virtual node* (vnode) represents access to an object within a virtual file system. Vnodes are used only to translate a path name into a generic node (gnode).

A vnode is either created or re-used for every reference made to a file by path name. Every time a user attempts to open or create a file, a list of existing vnodes is searched for the requested path name. (This list is accessed by consulting the **vfs** structure for the virtual file system that contains the vnode.) If a vnode already exists for the reference, a use count is incremented and the existing structure is used. Otherwise, a new vnode 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.
- The name of the virtual file system itself is not unique. An ancestor, including the mount point, can have multiple names.
- 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.)

Generic Inodes (Gnodes)

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

Each file system implementation is responsible for allocating and destroying gnodes. The gnode 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 gnode.

A gnode is needed, in addition to the file system inode, because some file system implementations may not include the concept of an inode. Thus the gnode 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 gnode:

gn_type	Identifies the type of object represented by the gnode.
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 vnode (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 vnode operations must supply a return value:
  - A return value of 0 (zero) indicates the operation was successful.
  - A nonzero return value is interpreted as a valid error number (taken from the **<sys/errno.h>** header 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 on invocation. The following are the necessary vfs operations:
  - **vfs_cntl**
  - **vfs_mount**
  - **vfs_root**
  - **vfs_statfs**
  - **vfs_sync**
  - **vfs_unmount**
  - **vfs_vget**.

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 vnode. 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 vnode.

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

vfs_flag Contains the state flags:

VFS_DEVMOUNT	Indicates whether the virtual file system has a physical mount structure underlying it.
VFS_READONLY	Indicates whether the virtual file system is mounted read-only.

vfs_lock Specifies the type of lock. A lock is either shared or exclusive. During path-name resolution, the virtual file system for the current vnode is share-locked. During mounting and unmounting, the virtual file system is

	exclusive-locked. To avoid deadlock, a vnode is *held* (using the **vn_hold** routine) when crossing a mount point and exclusive-locked during mounting and unmounting.
vfs_type	Identifies the type of file system implementation. Possible values for this field are described in the **<sys/vmount.h>** header file.
vfs_ops	Points to the set of operations for the specified file system type.
vfs_mntdover	Points to the mounted-over vnode.
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.

Data Structures and Header Files for Virtual File Systems

These are 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 vnodes 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 gnode 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, the **gfsadd** and **gfsdel** kernel services, 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 then operational.

Related Information

The **mount** subroutine, **mntctl** subroutine, **sysconfig** subroutine.
The **gfsadd** kernel service, **gfsdel** kernel service.
The Logical File System Kernel Services on page 6–12.

Kernel Services

Kernel services are routines that provide the runtime kernel environment to programs executing in kernel mode. Kernel services resemble library routines but are called by kernel extensions. Library routines are called by application programs.

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

The Kernel Services Alphabetical Listing in Appendix A. provides access to individual services, as well as list the execution environment (process or interrupt) from which the service can be called. System Calls Available to Kernel Extensions on page 4–7 lists the systems calls that kernel extensions can call.The following are the 12 categories of kernel services:

- Device Queue and Ring Queue Management Kernel Services
- I/O Kernel Services
- Kernel Program/Device Driver Management Kernel Services
- Logical File System Kernel Services
- Memory Kernel Services
- Message Queue Services Available from the Kernel
- Network Kernel Services
- Process and Exception Management Kernel Services
- RAS Kernel Services
- Security Kernel Services
- Timer and Time-of-Day Kernel Services
- Virtual File System Kernel Services.

Device Queue and Ring Queue Management Kernel Services

The Device Queue and Ring Queue services aid in porting previous AIX/VRM device drivers to the AIX Version 3 operating system. These device queues can help maintain the overall structure of an existing set of device managers that were implemented as kernel processes within VRM on previous versions of the AIX operating system.

The Device Queue services have, however, been streamlined and functionally reduced. Unlike the VRM device queue, these services are not part of the base operating system. They must be loaded into the kernel before use.

These services do not support the queued device driver model that existed in previous versions of the AIX operating system. These services support only process communications. The support of virtual interrupt handlers has also been changed. They now call the virtual interrupt-handling routines directly since virtual interrupts as such are no longer supported on the AIX operating system.

Note: It is strongly recommended that these functions be used only where quick, fairly straightforward porting of existing VRM device managers is required. Do not rely on either the Device Queue and Ring Queue services as a long-term model for kernel process communications. Other device management services and communications mechanisms should be investigated when developing new subsystems or when a considerable investment is required in porting previously implemented software.

Understanding Device Queues

Device queues are provided for compatibility with previous AIX device drivers. Device queues in AIX can be viewed as a method of queuing requests to a kernel process performing the device manager role. Kernel processes can use device queues to communicate to other kernel processes. For more information on kernel processes refer to Using Kernel Processes.

Loading Device Queue Management

Device queues are supported as an AIX kernel extension and must be loaded into the kernel before loading any other kernel extension that references them. The Device Queue Management kernel extension must only be loaded into the kernel once, which can be accomplished by loading with the SYS_SINGLELOAD **sysconfig** subroutine operation.

The Client/Server Model

Device queues are based on a client/server model. The *client* sends requests to the server and the *server* processes these requests.

A device queue server is a kernel process. The server consists of one or more routines providing the service. A device queue server that is part of a complex I/O subsystem is referred to as a *device manager.* Sophisticated device subsystems, such as those that involve virtualized devices, may require a device manager. For example, the SNA subsystem consists of several layers that interface to multiple devices including Ethernet and Token-Ring adapters. The device handlers are typically multiplexed character drivers that provide an interface to a logical link control layer. This control layer is most often a kernel process acting as a device manager.

To use a server's device queue a client must first attach to the device queue with the **attchq** Queue Management service. This creates a *path* from the client to the device queue and determines how acknowledgments are processed. Once a path is created, a client can send queue elements to the server with the **enque** Queue Management service. When finished, the client deletes the path with the **detchq** Queue Management service.

Queue Elements

There are two basic types of queue elements supported by device queues. A queue element sent from the client to the server is referred to as a *request* and is sent with the **enque** service. A queue element sent from the server to the client is referred to as an *acknowledgment* and is typically sent with the **deque** Queue Management service. However, acknowledgments can also be sent with the **ackque** Queue Management service. The path that connects the client to the server describes how both of these queue element types are processed.

Device queues can support more than one *queue element priority.* Priorities with numerically lower values are more favored than those with numerically higher values. More favored queue elements are processed before less favored queue elements. Queue elements with the same priority are processed in their arrival order.

There are two states in the life of a queue element. The queue element can be either *active* or *pending.* There is at most one active queue element for any device queue. A queue element is always placed in the device queue in the pending state. The queue element is then changed from the pending to the active state when the process either waits for a queue element (with the **waitq** Queue Management service) or reads the device queue (with the **readq** Queue Management service). The active queue element is removed from the device queue with the **deque** service. Thus the server directly controls the state of each queue element in the device queue.

Kernel processes serving device queues typically provide the interface to a logical or virtual device. This type of server usually consists of a kernel process coded as a loop. In this loop the process waits for a request, performs the operation specified by the request, acknowledges the request, and then repeats the loop. A process can serve more than one device queue. The **e_wait** kernel service can be used to wait on one or more device queues.

Device Queue Management Kernel Services

The Device Queue Management kernel extension provides the following 19 device queue management kernel services to support kernel processes using device queues. This kernel extension must be loaded by the first user before queue management services can be used.

Service	**Purpose**
ackque	Sends an acknowledgment queue element.
attchq	Creates a path to a device queue.
canclq	Deletes all pending queue elements from a device queue.
creatd	Assigns a global name to a device queue.
creatq	Creates a device queue.
deque	Performs completion processing for the active queue element.
detchq	Invalidates the path to a device queue.
dstryd	Deletes a global name from a device queue.
dstryq	Destroys the specified device queue.
enque	Sends a request queue element.
peekq	Returns a pending queue element in the device queue.
qryds	Returns information about the device manager.
queryd	Returns the device identifier associated with the IODN.
queryi	Provides information about device queues.
queryp	Indicates whether a path exists to a device queue.
readq	Returns the active queue element in the device queue.
vec_clear	Removes a registered virtual interrupt handler.
vec_init	Registers a virtual interrupt handler.
waitq	Waits for a queue element.

Warning: The device queue services assert that the object defined by an identifier is valid. The assert causes the system to crash. Therefore, the caller must ensure that the object on which it is performing an operation exists. This usually means that the caller needs to sequence its process termination carefully.

Device queue management directly calls some routines provided by the user of queue management under certain conditions.The function pointers for these routines are provided when using the **creatd** and **creatq** queue management services. The following routines can be directly called by queue management services:

- The **attach** routine initializes the server so that queued operations can be performed.
- The **detach** routine cleans up the server after the completion of all queued operations.
- The **check-parameters** routine verifies that each **enque** operation is valid before the operation is initiated.
- The **cancel-queue-element** routine cleans up resources associated with a queue element.
- The **virtual-interrupt-handler** routine handles virtual interrupts that are associated with the acknowledgement of a queue element.

Understanding Ring Queue Kernel Services

The 4 ring queue kernel services are also considered part of the device queue management services. They can be used in the process environment only.

The ring queue kernel services are a light-weight method of interprocess communication. They are typically used as part of a complex I/O subsystem and permit a device driver to notify and pass data to a kernel process efficiently. Ring queue services are typically used in communications device handlers to notify a kernel process of the availability of received data. As part of the notification process, they also pass the address of the buffer containing the received data.

The Ring Queue kernel services are:

rqc	Creates a ring queue.
rqd	Deletse a ring queue.
rqgetw	Gest the next element (word) from the ring queue.
rqputw	Puts a word into the next available element on the ring queue.

When the **rqputw** service places a word in an empty ring, a server process is notified with a specified event control bit or a server notification routine is called. The method of server notification is established by the *events* parameter on the call to the **rqc** service to create the ring queue. The function routine specified by the *func* parameter (also on the call to the **rqc** service) is called without parameters and has no return code.

I/O Kernel Services

The I/O kernel services fall into six categories: Block I/O services, Buffer Cache services, Character I/O services, Memory Buffer (mbuf) services, DMA Management services, and Interrupt Management services.

Block I/O Kernel Services

Block I/O device drivers are described in Understanding Block I/O Device Drivers. The three Block I/O kernel services are:

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.

Buffer Cache Kernel Services

The Block I/O Buffer Cache Kernel Services Overview describes how to manage the buffer cache with the Buffer Cache kernel services. The 14 Buffer Cache kernel services are:

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.
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.

Character I/O Kernel Services

Understanding Character I/O Device Drivers describes character device drivers. The 13 Character I/O kernel services are:

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.

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 **<sys/mbuf.h>** header file. Data can be stored directly in a memory buffer's data portion or in an attached external cluster. Memory buffers 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 15 Memory Buffer (mbuf) kernel services are:

m_adj	Adjusts the size of an **mbuf** chain.
m_cat	Appends one **mbuf** chain to the end of another.
m_clget	Allocates a page-sized **mbuf** structure cluster.
m_clgetx	Allocates an **mbuf** structure whose data is owned by someone else.
m_collapse	Guarantees that an **mbuf** chain contains no more than a given number of **mbuf** structures.
m_copy	Creates a copy of all or part of a list of **mbuf** structures.
m_copydata	Copies data from an **mbuf** chain to a specified buffer.
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_getclust	Allocates an **mbuf** structure from the **mbuf** buffer pool and attaches a page-sized cluster.

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:

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.
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.

DMA Management Kernel Services

The AIX operating system kernel provides 10 services for managing DMA channels and performing DMA operations. Understanding Direct Memory Access (DMA) describes DMA operations and channels.

d_align	Assists in allocation of DMA buffers.
d_clear	Frees a DMA channel.
d_complete	Cleans up after a DMA transfer.
d_init	Initializes a DMA channel.
d_mask	Disables a DMA channel
d_master	Initializes a block-mode DMA transfer for a DMA master.
d_move	Provides consistent access to system memory that is accessed asynchronously by a device and by the processor on a RISC System/6000.
d_roundup	Assists in allocation of DMA buffers.
d_slave	Initializes a block-mode DMA transfer for a DMA slave.
d_unmask	Enables a DMA channel.

DMA Transfers

A device driver must call the **d_slave** service to set up a DMA slave transfer or call the **d_master** service to set up a DMA master transfer. The device driver then sets up the device to perform the DMA transfer. The device transfers data when it is available and interrupts the processor upon completion of the DMA transfer. The device driver then calls the **d_complete** service to clean up after the DMA transfer. This process is typically repeated each time a DMA transfer is to occur.

Hiding DMA Data

In the RISC System/6000, data can be located in the processor cache, system memory, or a DMA buffer. The DMA services have been carefully written to ensure that data is moved between these three locations correctly. The **d_master** and **d_slave** services flush the data from the processor cache to system memory. They then hide the page, preventing data from being placed back into the processor cache. The hardware moves the data between system memory, the DMA buffers, and the device. The **d_complete** service flushes data from the DMA buffers to system memory and unhides the buffer.

A count is maintained of the number of times a page is hidden for DMA. A page is not actually hidden except when the count goes from 0 to 1 and is not unhidden except when the count goes from 1 to 0. Therefore, the users of the services must make sure to have the same number of calls to both the **d_master** and **d_complete** services. Otherwise, the page can be incorrectly unhidden and data lost. This count is intended to support operations such as logical volume mirrored writes.

Note: All pages containing user data must be hidden while DMA operations are being performed on them. This is required to ensure that data is not lost by being put in more than one of these locations.

DMA operations can be carefully performed on kernel data without hiding the pages containing the data. The **DMA_WRITE_ONLY** flag, when specified to the **d_master** service, causes it *not* to flush the processor cache or hide the pages. The same flag when specified to the **d_complete** service causes it *not* to unhide the pages. This flag requires that the caller has carefully flushed the processor cache using the **vm_cflush** service. Additionally, the caller must carefully allocate complete pages for the data buffer and carefully split them up into transfers. Transferred pages must each be aligned at the start of a DMA buffer boundary, and no other data can be in the same DMA buffers as the data to be transferred. The **d_align** and **d_roundup** services help ensure that the buffer allocation is correct.

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 allow buffers to be used for DMA operations to be aligned on a cache line boundary and allocated 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);
```

Note: If the kernel heap is used for DMA buffers, the buffer must be pinned using the **pin** kernel service before being used for DMA operations. Alternately, the memory could be requested from the pinned heap.

Accessing Data While the DMA Operation Is in Progress

Data must be carefully accessed when a DMA operation is in progress. The **d_move** service provides a means of accessing the data while a DMA transfer is being performed on it. This service accesses the data through the same system hardware as that used to perform the DMA transfer. The **d_move** service, therefore, cannot cause the data to become inconsistent. This service can also access data hidden from normal processor accesses.

Interrupt Management Kernel Services

Using Interrupts briefly describes the Interrupt Management kernel services and interrupt priorities. The eight Interrupt Management services are:

i_clear	Removes an interrupt handler.
i_disable	Disables interrupt priorities.
i_enable	Enables interrupt priorities.
i_init	Defines an interrupt handler.
i_mask	Disables a bus interrupt level.
i_reset	Resets a bus interrupt level.
i_sched	Schedules off-level processing.
i_unmask	Enables a bus interrupt level.

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 AIX file system for mounts and other limited activity, as well as for compatibility services required when other file systems are installed on AIX systems. These services are not used by the AIX JFS (Journal File System), NFS (Network File System), or CDRFS (CDROM File System) systems 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 reuse 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.

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 buffer cache 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 merely 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 there is doubt that 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 probably 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, while the **brelse**, **bwrite**, **bawrite**, or **bdwrite** services must eventually be called to free the block for use by other processes.

Other Buffer Cache Services

The following buffer cache kernel services also exist:

bflush	Flushes all write-behind blocks on the specified device from the buffer cache.
blkflush	Writes the data in a specified block to its device.
binval	Invalidates all of a specified device's data in the buffer cache.
clrbuf	Zeroes out the data buffer associated with a specified buffer header.
geterror	Returns the completion status of a buffer.
purblk	Invalidates a specified block's data in the buffer cache.

Understanding Interrupts

Each hardware interrupt has an interrupt level and an interrupt priority. The interrupt level defines the source of the interrupt. There are basically two types of interrupt levels: system and bus. The RISC System/6000 bus interrupts are generated from the Micro Channel bus and system I/O. Examples of system interrupts are the timer and serial link interrupts.

The interrupt level of a system interrupt is defined in the **<sys/intr.h>** header file. The interrupt level of a bus interrupt is one of the resources managed by the bus configuration methods.

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 degraded seriously.) 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 **<sys/intr.h>** header file. Processing Interrupts provides more information about interrupt levels and Early Power-Off Warning (EPOW) handlers.

Interrupt Services

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

i_init	Defines an interrupt handler to the system, connects it to an interrupt level, and assigns an interrupt priority to the level.
i_clear	Removes an interrupt handler from the system.
i_reset	Resets the system's hardware interrupt latches.
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.
i_enable	Enables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a more-favored interrupt priority.

Kernel Extension/Device Driver Management Kernel Services

The AIX kernel provides a relatively complete set of program and device driver management services. These services include general kernel extension loading and binding services and device driver binding services. Also provided are services that allow kernel extensions to be notified of base kernel configuration changes, user-mode exceptions, and systemwide process state changes.

Kernel Extension Loading and Binding Services

The **kmod_load**, **kmod_entrypt**, and **kmod_unload** services provide kernel extension loading and binding services. The **sysconfig** subroutine makes these services available to user-mode programs. However, kernel-mode callers executing in a kernel process environment can also use them. These services provide the same kernel object-file load, unload, and query functions provided by the **sysconfig** subroutine as well as the capability to obtain a module's entry point with the kernel module ID assigned to the module.

The **kmod_load**, **kmod_entrypt**, and **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. Device driver binding services include the **devswadd**, **devswdel**, and **devswqry** services, which are used to add or remove a device driver entry from the dynamically managed device switch table. They also query for information concerning a specific entry in the device switch table.

Other Functions for the Kernel Extension/Device Driver Management Services

Some kernel extensions may be sensitive to the settings of base kernel runtime configurable parameters that are found in the **var** structure defined in the **<sys/var.h>**, header file. These parameters can be set during system boot or runtime by a privileged user performing system configuration commands that use the **sysconfig** subroutine to alter values in the **var** structure. Kernel extensions may register or remove a configuration notification routine with the **cfgnadd** and **cfgndel** kernel services. This routine is called each time the **sysconfig** subroutine is used to change base kernel tunable parameters found in the **var** structure.

In addition, 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, being swapped in or swapped out. The **uexadd** and **uexdel** kernel services give kernel extensions the capability to intercept user-mode exceptions. These user-mode exception handlers may 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 **pioassist** and **getexcept** kernel services are typically 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 **selnotify** kernel service replaces the traditional UNIX **selwakeup** kernel function and is used by device drivers supporting the poll or select functions when asynchronous event notification is requested. 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.

Finally, the **getuerror** and **setuerror** services can be used by kernel extensions that provide or use system calls to access the **u.u_error** field for the current process. This is typically used by kernel extensions providing system calls to return error codes, and is used by other kernel extensions to check error codes upon return from a system call (since there is no **errno** global variable in the kernel).

The Kernel Extension/Device Driver Management Kernel Services

The 23 Kernel Program/Device Driver Management kernel services are:

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.
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 retrieve the current value of the **u_error** field.
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.
pio_assist	Provides a standardized programmed I/O exception handling mechanism for all routines performing programmed I/O.
prochadd	Adds a systemwide process state-change notification routine.
prochdel	Deletes a process state change notification routine.
selnotify	Wakes up processes waiting in a **poll** or **select** subroutine or the **fp_poll** kernel service.
setuerror	Allows kernel extensions to set the **u_error** field in the **u** area.
uexadd	Adds a systemwide exception handler for catching user-mode process exceptions.
uexblock	Makes a process non-runnable when called from a user-mode exception handler.
uexclear	Makes a process blocked by the **uexblock** service runnable again.
uexdel	Deletes a previously added systemwide user-mode exception handler.

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.

Other Considerations

The Logical File System services are available only in the 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.The following are the 17 Logical File System kernel services:

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 and/or channel number for a device.
fp_getf	Retrieves a pointer to a file structure.
fp_hold	Increments the open count for a specified file pointer.
fp_ioctl	Issues a control command to an open device or file.
fp_lseek	Changes the current offset in an open file.
fp_open	Opens a regular file or directory.
fp_opendev	Opens a device special file.
fp_poll	Checks the I/O status of multiple file pointers/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_rwuio	Performs read and 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_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.

Memory Kernel Services

The Memory kernel services provide kernel extensions with the ability to:

- 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.

The Memory kernel services are divided into the Memory Management services, Memory Pinning services, User Memory Access services, Virtual Memory Management services, and Cross Memory services.

The three Memory Management services are:

init_heap	Initializes a new heap to be used with kernel memory management services.
xmalloc	Allocates memory.
xmfree	Frees allocated memory.

The six Memory Pinning services are:

pin	Pins the address range in the system (kernel) space.
pincode	Pins the code and data associated with an object file.
pinu	Pins the specified address range in user or system memory.

unpin	Unpins the address range in system (kernel) address space.
unpincode	Unpins the code and data associated with an object file.
unpinu	Unpins the specified address range in user or system memory.

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 **copyin** or **copyout** services. The **uiomove** service is used for scatter/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 10 User Memory Access kernel services are:

copyin	Copies data between user and kernel memory.
copyinstr	Copies a character string (including the terminating NULL character) from user to kernel space.
copyout	Copies data between user and kernel memory.
fubyte	Fetches, or retrieves, a byte of data from user memory.
fuword	Fetches, or retrieves, a word of data from user memory.
subyte	Stores a byte of data in user memory.
suword	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.

Virtual Memory Management Kernel Services

These services are described in more detail in Understanding Virtual Memory Management Interfaces. The 22 Virtual Memory Management services are:

as_att	Selects, allocates, and maps a region in the specified address space for the specified virtual memory object.
as_det	Unmaps and deallocates a region in the specified address space that was mapped with the **as_att** kernel service.
getadsp	Obtains a pointer to the current process's address space structure for use with the **as_att** and **as_det** kernel services.
io_att	Selects, allocates, and maps a region in the current address space for I/O access.
io_det	Unmaps and deallocates the region in the current address space at the given address.
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_handle	Constructs a virtual memory handle for mapping a virtual memory object with specified access level.
vm_makep	Makes a page in client storage.
vm_mount	Adds a file system to the paging device table.
vm_move	Moves data between a virtual memory object and a buffer specified in the **uio** structure.
vm_protectp	Sets the page protection key for a page range.

vm_qmodify	Determines whether a mapped file has been changed.
vm_release	Releases virtual memory resources for the specified address range.
vm_releasep	Releases virtual memory resources for the specified page range.
vm_umount	Removes a file system from the paging device table.
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

Moving Data between Address Spaces

The cross memory 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 **xmemdma** service to prepare a page for DMA processing. The **xmemdma** service flushes any data from cache into memory and hides the page. A page is hidden by invalidating processor access to the page. Any processor references to the page result in page faults with the referencing process waiting on the page to be unhidden. The **xmemdma** service returns the real address of the page for use in preparing DMA address lists. When the DMA transfer is completed, the **xmemdma** 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 **pinu** service. This can only be done under a process, since the memory pinning services page-fault on pages not present in memory.

The **unpinu** 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 five Cross Memory services are:

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.
xmemdma	Prepares a page for DMA I/O or processes a page after DMA I/O is complete.

Virtual Memory Manager Interfaces

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

Several aspects of the virtual memory manager interface are discussed here:

- Virtual memory objects
- Addressing data
- Moving data to or from the kernel address space
- Moving data between address spaces
- Moving data to or from a virtual memory object
- Data flushing
- Protecting data
- Executable data
- Installing pager back ends.

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 map 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

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

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. This is an operation on the virtual memory object, not an address space range.

The **vm_move** kernel service moves 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 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 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 **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 to 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 instruction fetch does not need to use the data cache. The **vm_cflush** service performs this operation.

Installing Pager Back Ends

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 back ends.*

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 cannot sleep while 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.

Referenced routines

The virtual memory manager exports the following types of routines to kernel extensions:

- Services that manipulate memory objects
- Services that support address space operations
- Services that support the installation of pager back ends.

Services that support cross-memory operations are also included in the above group and are listed on page 6–15.

The following services 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_handle	Constructs a virtual memory handle for mapping a virtual memory object with a specified access level.
vms_iowait	Waits all page-outs for the virtual memory object to complete.
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_protectp	Sets the page protection key for a page range.
vm_releasep	Releases page frames and paging space slots for pages in the specified range.
vm_writep	Initiates page-out for a page range in a virtual memory object.

The following services support address space operations:

as_att	Selects, allocates, and maps a region in the specified address space for the specified virtual memory object.
as_det	Unmaps and deallocates a region in the specified address space that was mapped with the **as_att** kernel service.
getadsp	Obtains a pointer to the current process's address space structure for use with the **as_att** and **as_det** kernel services.
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 pageout for an address range.

Four Memory-Pinning kernel services also support address space operations. They are the **pin**, **pinu**, **unpin**, and **unpinu** services.

The following services support the installation of pager back ends:

vm_mount	Allocates an entry in the paging device table.
vm_umount	Removes a file system from the paging device table.

Message Queue Kernel Services Available from the Kernel

The Message Queue kernel services provide the same message queue functions to a kernel extension 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 IPC mechanism to other kernel processes or user-mode processes. For more information on functions provided by these services, see The Kernel Extension/Device Driver Management Kernel Services and The Device Queue and Ring Queue Management Kernel Services.

Message Queues Services Available from the Kernel

There are 4 Message Queue services available from the kernel:

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 are divided into the Address Family Domain and Network Interface Device Driver services, Routing and Interface services, Loopback services, Protocol services, and Communications Device Handler Interface services.

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 13 Address Family Domain and Network Interface Device Driver services are:

add_arp_iftype	Adds an interface type to the Network ARP Switch Table Interface (NASTI).
add_domain_af	Adds an address family to the Address Family domain switch table.
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_arp_iftype	Finds an interface type in the Network ARP Switch Table Interface (NASTI).
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 can use these to determine if an address is on a directly connected network. The 10 Routing and Interface Address services are:

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 two Loopback services are:

loifp	Returns the address of the software loopback interface structure.
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 four Protocol kernel services are:

pfctlinput	Invokes 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

The Communications Device Handler Interface services provide a standard interface between network interface drivers and AIX 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 5 Communications Device Handler Interface services are:

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 an AIX communications I/O device handler.
net_detach	Closes an AIX communications I/O device handler.
net_error	Handles errors for AIX communication network interface drivers.
net_sleep	Sleeps on the specified wait channel.
net_start	Starts network IDs on an AIX communications I/O device handler.
net_start_done	Starts the done notification handler for AIX communications I/O device handlers.
net_wakeup	Wakes up all sleepers waiting on the specified wait channel.
net_xmit	Transmits data using an AIX communications I/O device handler.

Process and Exception Management Kernel Services

The Process and Exception Management kernel services provided by the base AIX kernel provide the capability to:

- Create kernel processes
- Register exception handlers
- Provide process serialization
- Generate and handle signals
- Support event waiting and notification.

Kernel extensions can use the **creatp** and **initp** kernel services to create and intialize a kernel process. A kernel process can use the **sig_chk** kernel service to poll for signals that have been sent to the 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 signal upon kernel process termination. Using Kernel Processes supplies additional information concerning use of these services.

The **setjmpx**, **clrjmpx**, and **longjmpx** kernel services allow a kernel extension to register an exception handler by:

- Saving the exception handler's context with the **setjmpx** kernel service
- Removing its saved context with the **clrjmpx** kernel service if no exception occurred
- Invoking the next registered exception handler with the **longjmpx** kernel service if it was unable to handle the exception.

Refer to Handling Exceptions While in a System Call for additional information concerning use of these services.

The **lockl** and **unlockl** kernel services allow kernel extensions executing in the process environment to acquire or release locks that are typically used to serialize access to a resource. The **getpid** kernel service can be used by a kernel extension in either the process or interrupt environment to determine the current execution environment and obtain the process ID of the current process if in the process environment.

The event notification services provide support for primitive interprocess communications where there can be only one process waiting on the event or shared event interprocess communications where there can be multiple processes waiting on the event. The traditional **sleep** and **wakeup** kernel services are also provided for code that is being ported from other UNIX operating systems or previous versions of the AIX operating system. These compatibility services require that the caller have the global **kernel_lock**, which is released before waiting in the sleep routine and re-acquired upon wakeup.

The **e_wait** and **e_post** kernel services support single waiter event notification by using mutually agreed upon event control bits for the process 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 **e_wait** service, it is released and re-acquired upon wakeup.

The **e_wakeup**, **e_sleep** and **e_sleepl** kernel services support a shared event notification mechanism that allows for multiple processes to be waiting on the shared event. These services support an unlimited number of shared events (by using caller-supplied event words). All processes waiting on the shared event are awakened by the **e_wakeup** service. If the caller of the **e_sleep** service owns the **kernel lock**, it is released before waiting and is reacquired upon wakeup. The **e_sleepl** service provides the same function as the **e_sleep** service except that a caller-specified lock is released and reacquired instead of the **kernel_lock**.

The Process and Exception Management Kernel Services

There are 19 Process and Exception Management kernel services:

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.
e_post	Notifies a process of the occurrence of one or more events.
e_sleep	Forces a process to wait for the occurrence of a shared event.
e_sleepl	Forces a process to wait for the occurrence of a shared event.
e_wait	Forces a process to wait for the occurrence of an event.
e_wakeup	Notifies processes waiting on a shared event of the event's occurrence.
getpid	Gets the process ID of the current process.
initp	Changes the state of a kernel process from idle to ready.
lockl	Locks a conventional process lock.
longjmpx	Allows exception handling by causing execution to resume at the most recently saved context.
pdsignal	Sends a signal to a process group.

pidsig	Sends a signal to a process.
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 a kernel process the ability to poll for receipt of signals.
sleep	Forces the calling process to wait on a specified channel.
unlockl	Unlocks a conventional process lock.
wakeup	Activates processes sleeping on the specified channel.

RAS Kernel Services

The RAS kernel services are used to record the occurrence of hardware or software failures and to capture data about these failures. The recorded information can be examined using the **errpt**, **trcrpt**, or **crash** commands.

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_add** kernel service to add an entry to the Master Dump Table and the **dmp_del** kernel service to remove an entry.

The **errsave** 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 **trchk** subroutine to record selected system events in the event tracing facility.

The RAS Kernel Services

The 6 RAS kernel services are:

dmp_add	Specifies data to be included in a system dump by adding an entry to the master dump table.
dmp_del	Deletes an entry from the master dump table.
errsave	Allows the kernel and kernel extensions to write to the error log.
panic	Crashes the system.
trcgenk	Records a trace event for a generic trace channel.
trcgenkt	Records a trace event, including a time stamp, for a generic trace channel.

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 are the 4 Security kernel services:

audit_svcbcopy	Appends event information to the current audit event buffer.
audit_svcfinis	Writes an audit record for a kernel service.
audit_svcstart	Initiates an audit record for a system call.
suser	Determines the privilege state of a process.

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 AIX 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 Time-of-Day services, Fine Granularity Timer services, Timer services for compatibility, and Watchdog Timer services. There is only one Time-of-Day kernel service: the **curtime** kernel service.

The five Fine Granularity Timer kernel services are:

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.

You can find additional information about using the Fine Granularity Timer services in Using Fine Granularity Timer Services and Structures.

Three Timer kernel services are provided for compatibility:

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.
untimeout	Cancels a pending timer request.

The following are the 4 Watchdog kernel services:

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.
w_start	Starts a watchdog timer.
w_stop	Stops a watchdog timer.

Using Fine Granularity Timer Services and Structures

The **tstart**, **tfree**, **talloc**, and **tstop** services provide fine-resolution timing functions. These timer services should be used when the following conditions are required:

- Timing requests for less than one second
- Critical timing
- Absolute timing.

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>** header file. The **itimerstruc_t** structure contains the second/nanosecond structure for time operations and is found in the **<sys/time.h>** header file.

The **itimerstruc_t t.it** value substructure should be 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. It should be ORed into the *t*–**>flag** field if an absolute timer request is desired.

The *t*–**>timeout** and *t*–**> flags** fields must be set or reset before each call to the **tstart** kernel service.

Coding the Timer Function

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.

The *t*->**func** timer function is called on an interrupt level. Therefore, code for this routine must follow conventions for interrupt handlers.

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 vnodes, 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 six VFS kernel services are:

gfsadd	Adds a file system type to the **gfs** table.
gfsdel	Removes a file system type from the **gfs** table.
vn_free	Frees a vnode previously allocated by the **vn_get** kernel service.
vn_get	Allocates a virtual node and inserts it into the list of vnodes for the designated virtual file system.
vfsrele	Points to a virtual file system structure.
lookupvp	Retrieves the vnode that corresponds to the named path.

Related Information

The **sysconfig** subroutine.
The **e_wait** kernel service.
The **iostat** command, **vmstat** command.
Understanding Block I/O Device Drivers on page , Understanding Character I/O Device Drivers on page 3–6, Direct Memory Access (DMA) on page 3–15.
Device Driver Classes on page 2–1.
The **buf** structure in *Kernel Extensions and Device Support Programming Concepts* .
Processing Interrupts on page 3–12.
The Virtual File System Interface on page 5–4 .

The Configuration Subsystem

The following topics are discussed in the Device Configuration Subsystem:

- Scope of AIX Device Configuration Support
- Device Configuration Database Overview
- Device Classes, Subclasses, and Types
- Writing a Device Method
- Understanding System Boot Processing.

Scope of AIX Device Configuration Support

In the AIX device configuration subsystem, the term *device* has a wider range of meaning than it does in traditional UNIX systems.

In both AIX and UNIX systems, the term *devices* refers to hardware components such as disk drives, tape drives, printers, and keyboards. Pseudo-devices, such as the console, error, and the null special file, are also included. For AIX, all of these devices are referred to as the *kernel devices*, that is, the devices with device drivers and known to the system by major and minor numbers.

However, in the AIX operating system, hardware components such as buses, adapters, and even enclosures (including racks, drawers, and expansion boxes) are also considered devices.

The Devices Graph diagram on page 7–6 provides more information about the connections and dependencies between these components.

Note: The system cannot use any device unless it is configured.

General Structure of the Device Configuration Subsystem

The Device Configuration Subsystem can be viewed from three different levels: the High Level Perspective, the Device Method Level, and the Low Level Perspective. The Overview of System Management of Devices diagram on page 7–2 illustrates the general structure of the Device Configuration Subsystem.

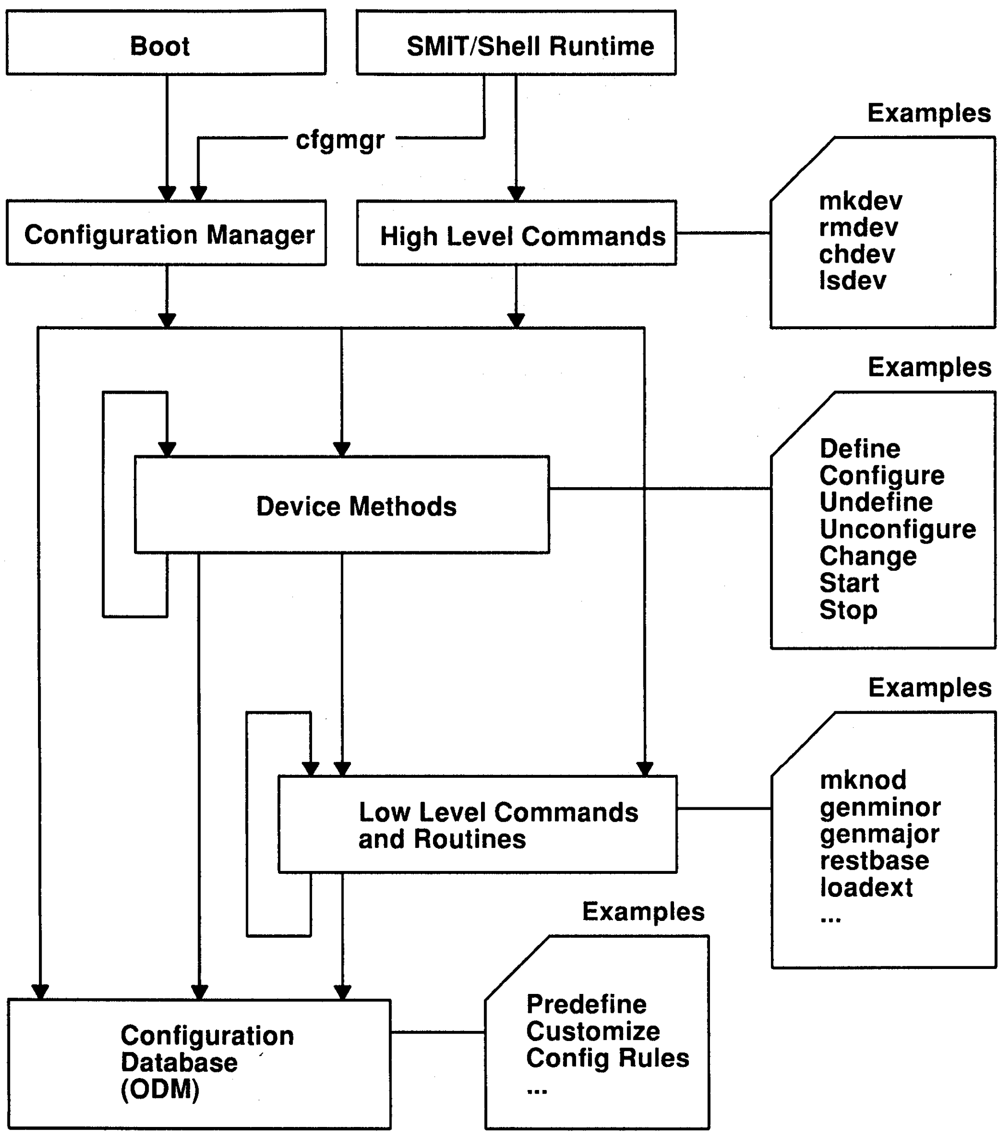
High Level Perspective

From a high level, user-oriented perspective, four basic tasks comprise device configuration:

- Adding a device to the system
- Deleting a device from the system
- Changing the attributes of a device
- Showing information about a device.

A set of high level commands accomplish these tasks during runtime: **chdev**, **mkdev**, **lsattr**, **lsconn**, **lsdev**, **lsparent**, **rmdev**, and **cfgmgr**.

The *Configuration Database* stores all information relevant to support the device configuration process. It 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 and configured in that particular system.



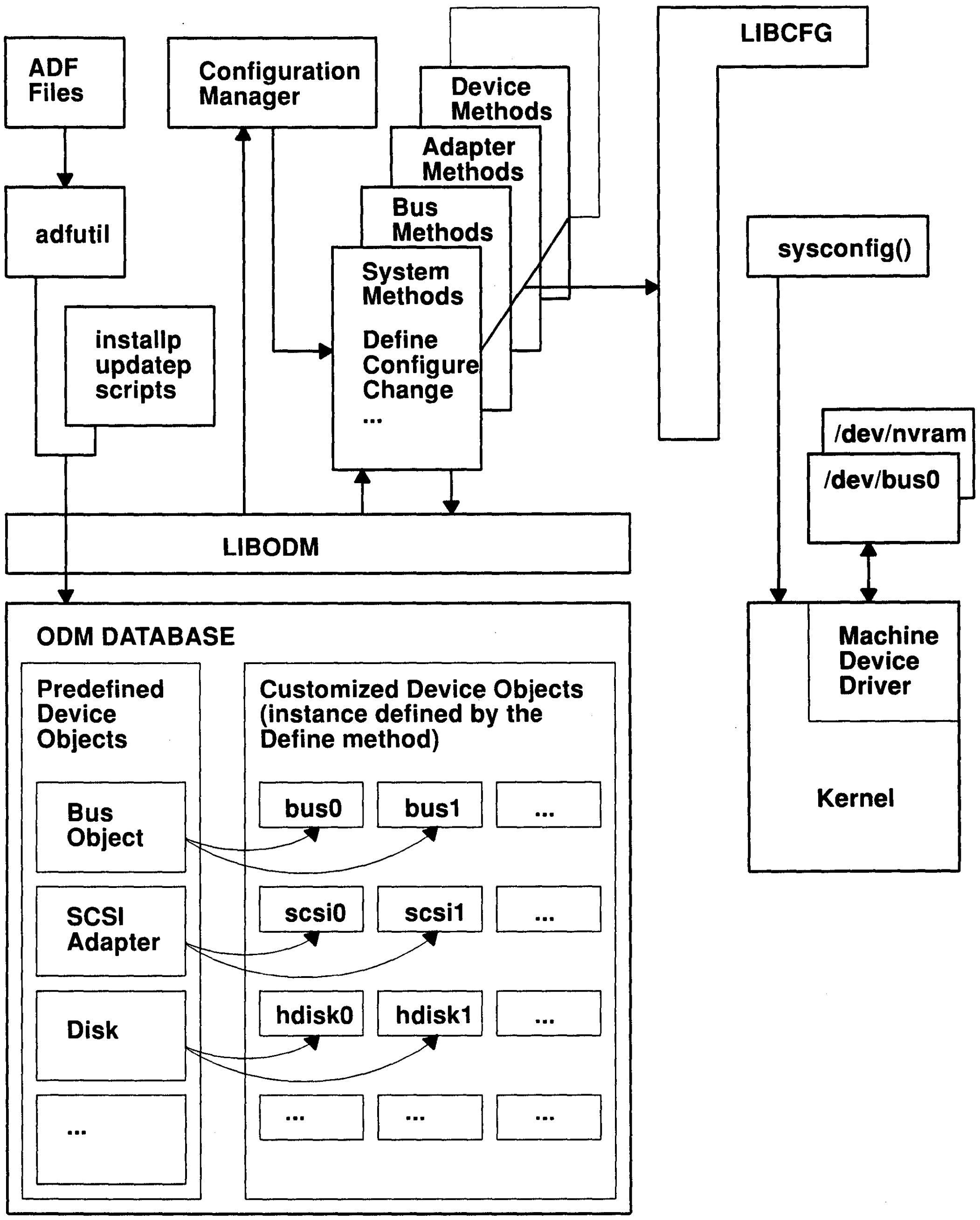
Overview of System Management of Devices

The *Configuration Manager* supervises the configuration of a system's devices when the system is booted. These components are illustrated in the Device Configuration Support diagram on page 7–3.

Device Method Level

Beneath the high-level devices commands or the Configuration Manager 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.



Components Involved in Device Configuration Support

Device methods also provide two optional functions for devices that need them:

- Starting a Device to take it from the Stopped state to the Available state
- Stopping a Device to take it to the Stopped state.

Understanding Device States contains a diagram illustrating all possible device states and how the various methods effect device state changes.

Both the high-level device commands and the Configuration Manager can use the device methods. These methods are implemented to insulate higher level configuration programs from kernel-specific, hardware-specific, and device-specific configuration steps.

Low Level Perspective

Beneath the device methods is a set of low-level device configuration commands and library routines that can be directly called by device methods as well as by higher level configuration programs.

Device Configuration Database Overview

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.

The Configuration Database is an object-oriented database. The Object Data Manager (ODM) provides facilities for accessing and manipulating it.

There are actually two databases used in the configuration process:

Predefined Database	Contains information about all possible types of devices that can be defined for the system.
Customized Database	Describes all devices currently defined in the system.

ODM Device Configuration Object Classes provides access to the object classes that make up the Predefined and Customized databases.

Basic Device Configuration Procedures

At system boot time, the Configuration Manager is automatically invoked to configure all devices detected and devices whose device information is stored in the Configuration Database. At runtime, you can configure a specific device by directly invoking, or indirectly invoking through a usability interface layer, high-level devices commands. This interface is illustrated by the General Structure of the Configuration Subsystem diagram.

High-level devices 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 needs to:

- Load the device's driver into the kernel
- Pass the Device-Dependent structure (DDS) describing the device instance to the driver
- Create a special file for the device in the /**dev** directory.

Of course, 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 begins with the System device shown in the Devices Graph figure on page 7–6 and configures it. The remaining devices are then 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

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.

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 are the adapters, which are connected to the bus. The bottom of the hierarchy contains the devices to which no other devices are connected. Most of the pseudo-devices, including HFT and PTY, are organized as separate tree structures or nodes.

The Devices Graph

The Devices Graph diagram on page 7–6 provides an example of connections and dependencies of devices in the system. Device Dependencies and Child Devices on page 7–12 provides 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 top of the nodes. In invoking these programs, the names of the next lower level devices that need to be configured are returned.

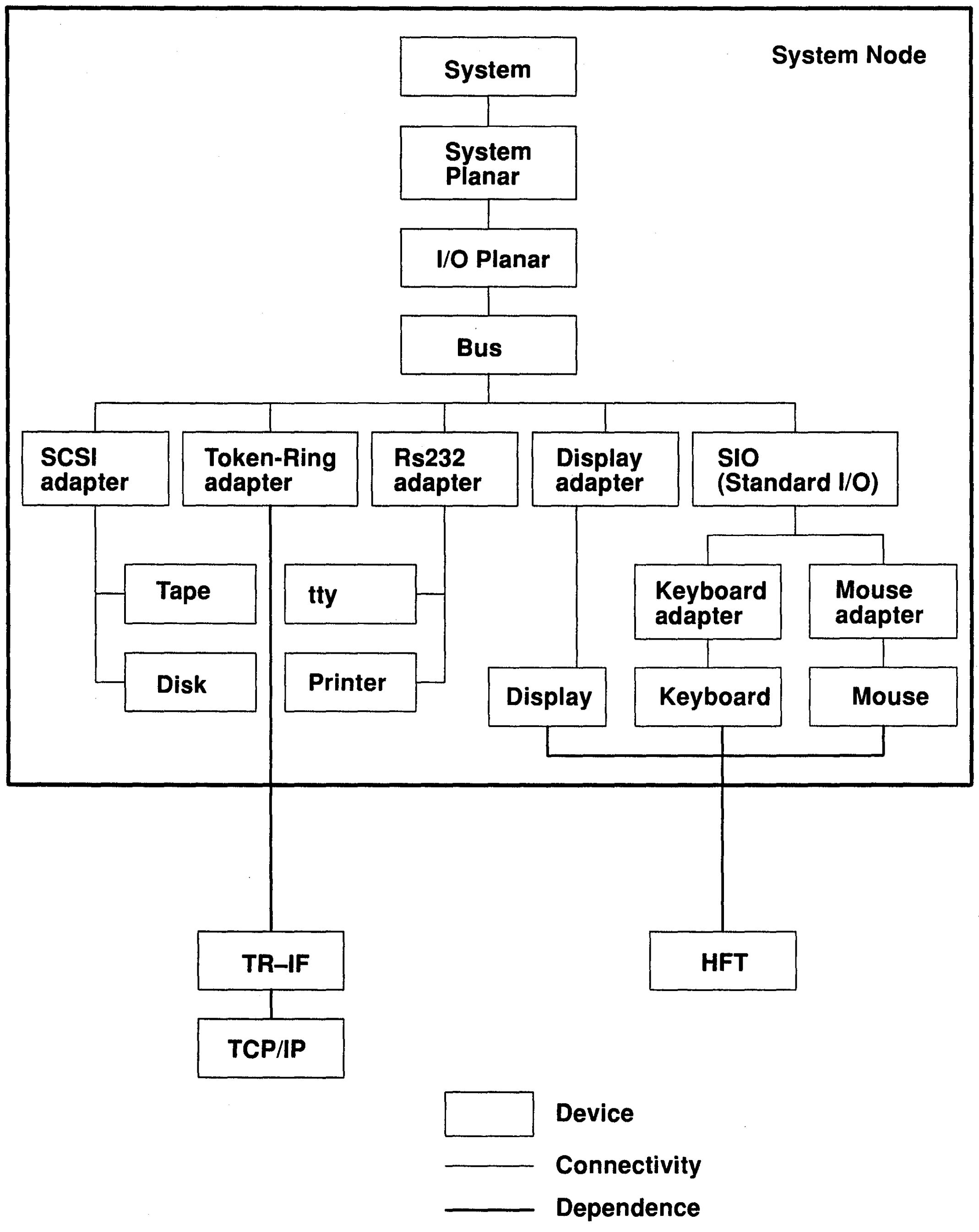
The Configuration Manager configures the next lower level devices by invoking the configuration methods for those devices. In turn, those configuration methods return a list of to-be-configured device names. The process is repeated until no more device names are returned. As a result, all devices in the same node are configured in transverse order. There are three different types of rules: phase 1, phase 2, and 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** option, which specifies that the *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** option, which specifies that the *phase = 2* rules are to be executed. This results in the configuration of the rest of the devices into the system.

The Understanding System Boot Processing on page 7–14 contains diagrams that illustrate the separate step of system boot processing.

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 sequence number of 2 is executed before a rule with a sequence number of 5. An exception is made for sequence numbers of 0, 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 devices 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 may not be associated with any devices, but they must be done to configure the system.



Devices Graph: Examples of Connectivity and Dependence

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-up 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 the key switch position on the front panel. If the key position is in service position, the rules for service mode are used. Otherwise, the phase 2 rules are used.

The Configuration Manager can also be invoked during runtime to configure all the detectable devices that may 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

To manage its wide variety of devices more easily, the AIX 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 these three 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* where devices belonging to the same subclass have similar interfaces. For example, serial printers and parallel printers form two subclasses of the class 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/or number. For example, IBM 3812–2 (model 2 Pageprinter) and IBM 4201 (Proprinter II) printers comprise 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 IBM 4201 printer belongs to both the serial interface and parallel interface subclasses of the printer class, and there are different drivers for the two interfaces. But 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. The top of the node is the System device. Below the bus are the adapters, which are connected to the bus. The bottom of the hierarchy contains the devices to which no other devices are connected. Most of the pseudo-devices, including HFT and PTY, are organized as separate nodes.

The Devices Graph on page 7–6 illustrates this structure.

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.

There are five basic device methods:

Define	Creates a device instance in the Customized database.

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 an 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.

Some devices also require these two optional methods:

Stop	The Stop method provides the ability to stop a device without actually unconfiguring it. For example, a command can be issued to the device driver telling it to stop accepting normal I/O requests.
Start	This method is used to start a device that has been stopped with the stop method. This may simply be a command to the device driver informing it that it can now accept normal I/O requests.

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 may invoke the Unconfigure and Configure methods. To ensure proper operation, a method that invokes another method must always use the **odm_run_method** subroutine.

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, it is important for device methods to 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 are required to 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.

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. 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 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 children need to be configured. If so, the Configure method writes the names of the 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 operate as described for the parent device. For example, they might simply exit when complete, or write to their **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.

The Runtime Configuration Commands

- The **mkdev** Command

  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.

  If defining and configuring a device, the **mkdev** command invokes the Define method, gets the name written to the **stdout** file by the Define method, invokes the Change method for the device if user-specified attributes were supplied, and finally invokes the device's Configure method.

  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.

- The **chdev** Command

  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.

- The **rmdev** Command

  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.

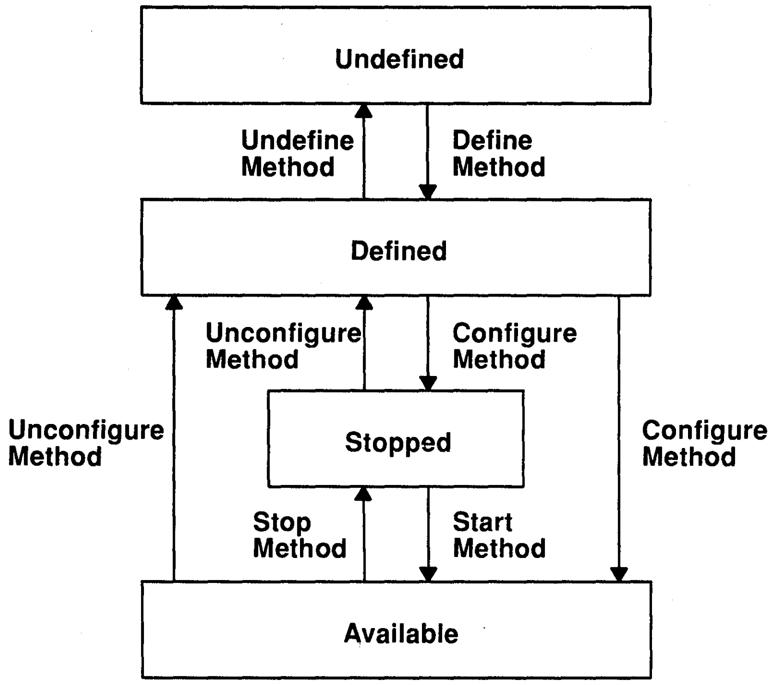
- The **cfgmgr** Command

  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 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.

The Device States diagram illustrates the possible states and the device methods that affect them.



Device States

Defined	Represented in the Customized database, but is not configured and not available for use in the system.
Available	Configured and available for use.
Undefined	Not represented in the Customized database.
Stopped	Configured, but not available for use by applications. (Optional state)

The Define method is responsible for creating the 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 Available, the Change method attempts to apply the changes to both the database and the actual device and again leave the device in the same state. However, if an error occurs when applying the changes to the actual device, the Change method may need to unconfigure the device, thus changing the state to `defined`.

The Unconfigure method must perform the operations necessary to make the device no longer usable. Basically, this is to undo the operations performed by the Configure method. It will set 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 may need to use. 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`.

Adding an Unsupported Device to the System

The AIX operating system provides support for a wide variety of devices that can be added to your system. 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.

To add a currently unsupported device to your system, you may need to:

- Modify the Predefined database
- Write appropriate device methods
- Write a device driver, in some cases
- Use **installp** procedures.

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** ODM (Object Data Manager) command from the command line or in a shell script to populate the necessary Predefined object classes from stanza files.

The Predefined database is shipped prepopulated for IBM-supported devices. For some IBM-supported devices such as serial and parallel printers and SCSI disks, the database has also been prepopulated with generic device objects. These generic device objects can be used to configure other similar devices that are not explicitly supported in the Predefined database.

For example, if you have a serial printer that closely resembles a printer supported by the system, and you believe that the system's device driver for serial printers will work for your printer, you can add the device driver in as a printer of type `osp` (other serial printer). If these generic devices work for adding your device, you do not need to provide any additional system software.

Adding Device Methods

You must add device methods when adding system support for a new device. Five methods are needed to support a device: the Define, Configure, Change, Undefine, and Unconfigure methods.

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 adding a new type of SCSI disk whose interfaces are identical to supported SCSI disks, the existing methods for SCSI disks may work for you. If so, you only need to do is to populate the Predefined database with information describing the new SCSI disk similar to the information describing a supported SCSI disk.

If you need instructions on how to write a device method, see Writing a Device Method Overview.

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 adding a new type of SCSI disk whose interfaces are identical to supported SCSI disks, the existing SCSI disk device driver may work for you. Device Driver Kernel Extension Overview offers guidelines for writing your own device driver.

Using installp Procedures

The **installp** procedures provide a method of adding the software and Predefined information needed to support your new device. You may need to write shell scripts to perform tasks such as populating the Predefined database. Program Installation and Update Compatibility Overview has more information on using **installp** procedures.

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) objects.

A device method can also add to the Customized Dependency (CuDep) object class an object identifying both a dependent device and the device upon which it depends. The dependent device is considered to *have* a dependency, and the depended-upon is considered to *be* a dependency. Customized Dependency objects are usually added to the database to represent a situation in which one device requires access to another device. For example, the hft0 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 retrieve the names of the devices on which it depends. The Configure method can then check to see if those devices exist.

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 which the child's device driver may be using remains valid.

However, when a device is listed as a dependency for 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 be assigned the same name.

Writers of Unconfigure and Change methods for a depended-upon device need 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 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. Subclass is used to identify the devices since it indicates the devices's connection type (for example, SCSI or RS-232).

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 PdAt object class.

The Devices Graph diagram on page 7–6 in provides an example of device dependencies and connections in the system.

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 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 will be 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, your methods must also obtain the objects for that attribute from both the PdAt and CuAt object classes.

Here are four scenarios that your methods must be able to handle when modifying attribute values:

1. If the new value differs from the default value and no object currently exists in the CuAt object class, your method must add an object into the CuAt object class to identify the new value.
2. If the new value differs from the default value and an object already exists in the CuAt object class, your method must update the CuAt object with the new value.
3. If the new value is the same as the default value and an object exists in the CuAt object class, your method must delete the CuAt object for the attribute.
4. If the new value is the same as the default value and no object exists in the CuAt object class, your method 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.

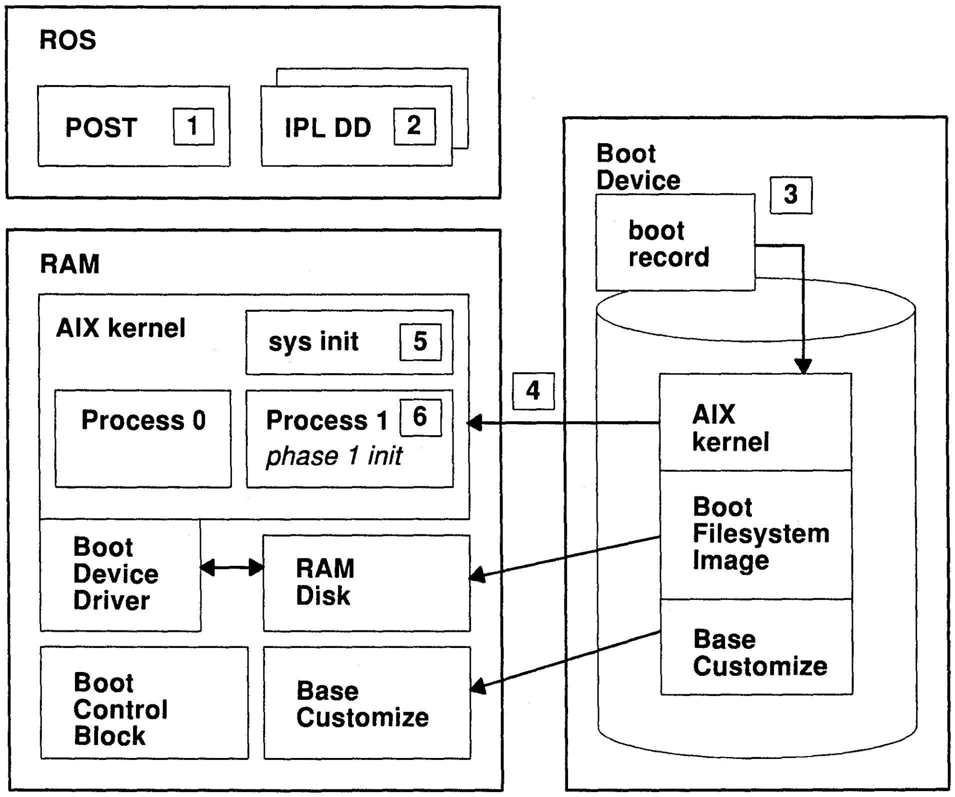
To help in modifying attributes, the **putattr** routine handles these four cases.

Understanding System Boot Processing

The ROS/Kernel Init Phase diagram illustrates kernel initialization that takes place before Phase 1 of the system boot is started.

System Boot

ROS/Kernel Init Phase



In the illustration of the ROS/Kernel Init Phase on page 7–14 above, the machine begins executing in ROS and does the following:

1. Checks that the machine is operation (POST).
2. Locates the Boot Device, which is platform specific.
3. Reads the Boot Record and locates the kernel and Boot Filesystem image.
4. Loads the AIX kernel, boot filesystem and base customized information into RAM.

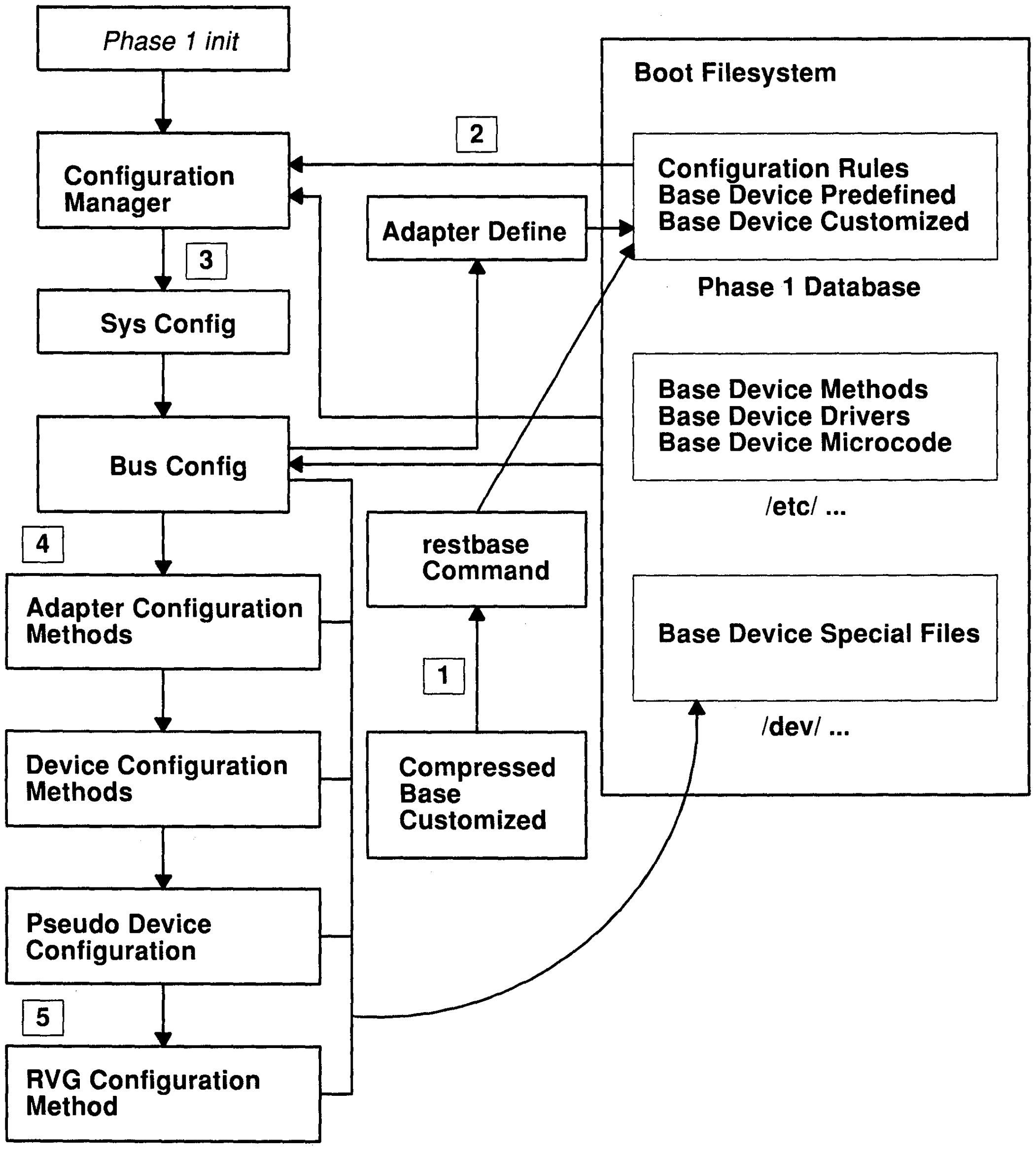
The Kernel gets control from ROS with a pointer to the boot control block and the following occurs:

1. System initialization starts.
2. Process 1 executes *phase 1 init.*

The Phase 1: Base Device Configuration diagram below illustrates the steps of system boot Phase 1.

System Boot

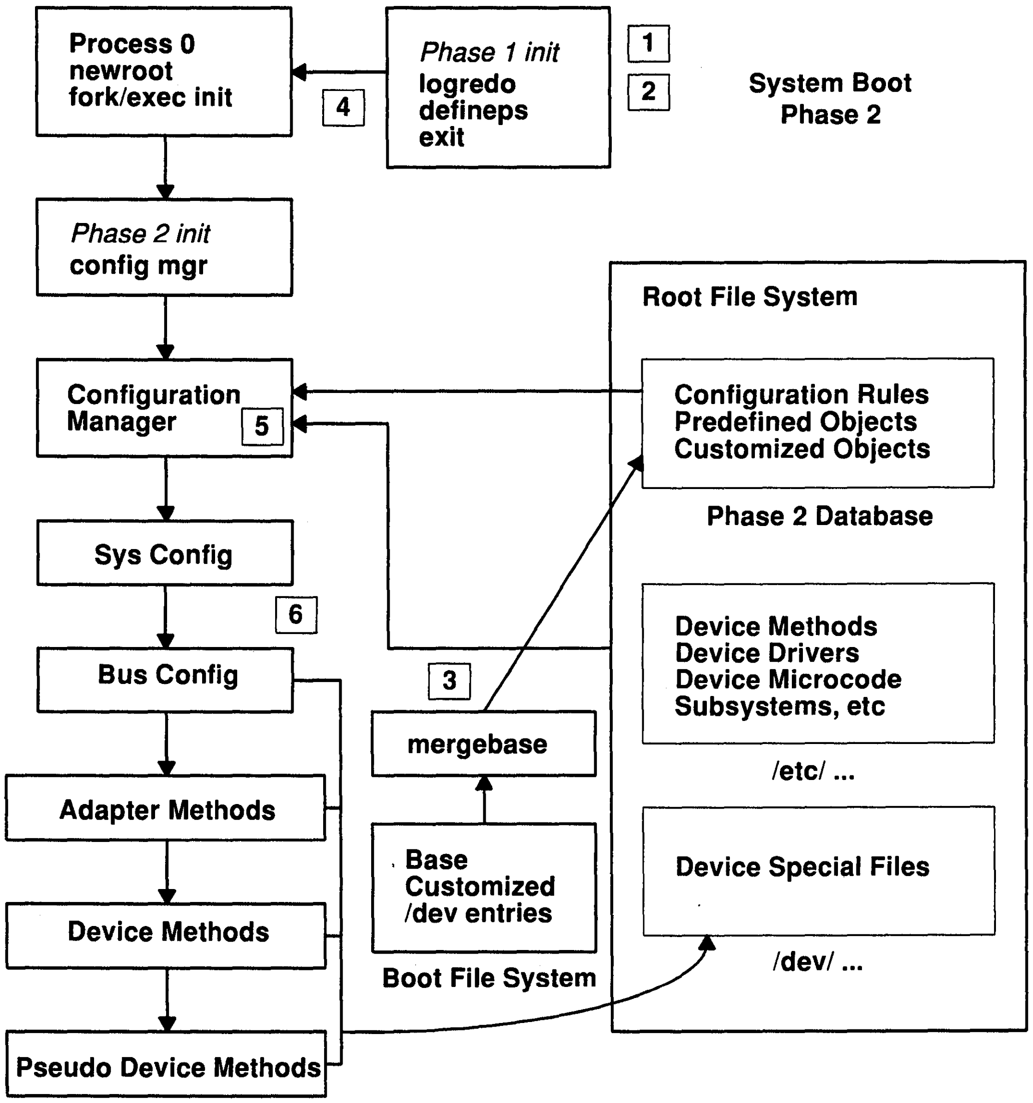
Phase 1: Base Device Configuration



In the illustration on page 7–15 above, the Init executes the configuration manager and the following occurs:

1. The customized database is built from the packed base customize data.
2. The Phase 1 Configuration Rules are accessed to run methods.
3. The Sys, Bus, SCSI, Disk, Tape, Diskette, LVM/RVG, tty, and HFT methods are executed.
4. The config methods load drivers, make the special files, and update customize data.
5. The Root Volume group is varied on (normal boot).

Phase 2 diagram illustrates the steps of system boot Phase 2.



In the illustration above the Normal Boot, Install/Maintenance, or Diagnostics are executed and the following 6 actions occur:

1. A Logredo is run on the Root Filesystem.
2. The pager is started.
3. The Base Customized is added to the Phase 2 data base, and the Phase is started.

4. A New root is performed to switch from the Boot filesystem to the Root filesystem.
5. The Config Manager is started for Phase 2 (configure using full predefined, customized database).
6. All devices/subsystems not defined/configured in phase 1 are defined and configured as required.

Related Information

The **odm_run_method** subroutine.

The **cfgmgr** command, **mkdev** command, **chdev** command, and **rmdev** command.

The **odmadd** command, **installp** command.

Writing a Define Method, Writing Optional Start and Stop Methods, Writing a Define Method, Writing a Undefine Method, Writing a Unconfigure Method, Writing a Configure Method, Writing Change Method in *Calls and Subroutines Reference.*

Configuration Rules (Config_Rules) object class, Customized Devices (CuDv) object class, Customized Dependency (CuDep) object class,Predefined Connection (PdCn) object class, Predefined Attribute (PdAt) object class, Predefined Devices (PdDv) object class, Customized Attributes(CuAt) object class, ODM Device Configuration Object Classes.

Device Methods For Adapter Cards, Handling Device Vital Product Data (VPD), Loading a Device Driver, Device Configuration Subroutines and Commands, Understanding Configure Method Errors, Machine Device Driver, in *Calls and Subroutines Reference*

Special File Overview in *AIX General Programming Concepts for IBM RISC System/6000.*

Device Dependent Structure (DDS) Overview on page 2–3.

Kernel Extension Binding on page 1–2.

Pseudo-Device Drivers on page 3–10.

Kernel Environment Programming on page 1–1.

The 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.

A communications PDH is a special type of multiplexed character device driver. Information common to all the communications device handlers is discussed here. However, individual communications PDHs have their own adapter-specific sets of information:

- Ethernet adapter
- Multiprotocol adapter
- Token-Ring adapter
- X.25 adapter.

Each adapter type requires a device driver. Each PDH can support one or more adapters of the same type.

User–Mode Interface to a Communications PDH

The user-mode process uses system calls to interface to the PDH to send or receive data. The **poll** or **select** subroutine is used to notify 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 these two 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** operation, Get Status **ddioctl** (CIO_GET_STAT) operation, and **ddselect** 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 a PDH's priority.

Communications Physical Device Handler Model Overview

A physical device handler (PDH) must be provided eight common entry points. An individual PDH names its entry points by placing a two- or three-letter unique designation in front of the command type supported. For example, the Ethernet open is named the *ent***open** entry point. The following are the required eight communications PDH entry points:

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 that the common **ddmpx** device handler entry point is.
ddopen	Performs data structure allocation and initialization for a communications PDH. Supported the same way that the common **ddopen** entry point is. Time-consuming tasks, such as port initialization and connection establishment, are deferred until the **ddioctl** (CIO_START) 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 that the common **ddclose** entry point is.
ddwrite	Queues a message for transmission or blocks until the message can be queued. The **ddwrite** entry 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. Supported the same way that the common **ddselect** entry point is.
ddioctl	Performs the special I/O operations requested in an **ioctl** subroutine. Supported the same way that the common **ddioctl** entry point is. In addition, a communications PDH must support the following four additional options: • CIO_START • CIO_HALT • CIO_QUERY • 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 PHD

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 **<sys/mbuf.h>** file.

The **m_next** field is used to chain **mbuf** structures together on linked lists. The **m_act** field allows lists of **mbuf** chains to be accumulated. By convention, the **mbuf** structures common to a single object are chained together with the **m_next** field, while groups of objects are linked together with the **m_act** field. Chains of **mbuf** structures tied together with the **m_act** field are not supported.

Each **mbuf** structure has a small data area for storing information: the **m_dat** area. The **m_len** field indicates the amount of data, while the **m_off** field is an offset to the beginning of the data from the base address of the **mbuf** structure.

Rather than storing data directly in the **mbuf** data area, data can instead be stored in a separate mbuf cluster. When the **m_off** field is less than 0 or greater than size indicated by the **MSIZE** value, then the data is in the mbuf-cluster buffer.

Chained **m_buf** structures can contain an **m_len** field of 0. If 0, the buffer is ignored. The chain must contain enough data to create a packet of minimum size.

Common Communications Status/Exception Codes

In general, communication device-handlers return codes from a group of common exception codes. However, device handlers for specific communication devices may return device-specific exception codes. Common exception codes are defined in the **<sys/comio.h>** file and include these eleven codes:

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	Enter 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.
CIO_LOST_STATUS	Indicates that a status block was lost.
CIO_NETID_INV	Indicates that the network ID was invalid.
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.

Ethernet Device Handler

The Ethernet device handler is a component of the Communication I/O Subsystem. The Ethernet device handler can handle one to four adapters simultaneously. It consists of the following eight entry points:

entconfig	Provides functions for initializing, terminating, and querying the vital product data (VPD) of the Ethernet device handler.
entopen	Initializes the Ethernet device handler and allocates the required system resources.
entclose	Resets the Ethernet device to a known state and returns system resources back to the system.
entmpx	Provides allocation and deallocation of a channel.
entwrite	Provides the means for transmitting data from the Ethernet device.
entread	Provides the means for receiving data from the Ethernet device.
entselect	Determines if a specified event has occurred on the Ethernet device.
entioctl	Provides various functions for controlling the Ethernet device. The **entioctl** entry points provides the following five common operations and an additional Ethernet specific operation:

IOCINFO	Returns some I/O character information.

CIO_START	Starts a session and registers a network ID.
CIO_HALT	Halts a session and removes a registered network ID.
CIO_QUERY	Returns the current RAS Counter values.
CIO_GET_STAT	Returns current adapter and device handler status.
CCC_GET_VPD	Returns VPD about the Ethernet adapter.
ENT_SET_MULTI	Sets or removes a multicast address.

Note: The **ent_stats** structure defines the RAS Log structure for the Ethernet device handler and associated labels. The structure's counters are initialized to 0 (zero) on the first **entopen** entry point. The **ent_stats** structure is found in the **<sys/entuser.h>** file.

Data Transmission for the Ethernet Device Handler

Data transmission for the Ethernet device handler is dependent on the type of process. To transmit data in user-mode, an application issues a **write**, **writev**, **writex**, or **writevx** subroutine and specifies a buffer address. The Ethernet device handler copies the buffer to a kernel space **mbuf** structure. The device handler then gives the Ethernet adapter the packet for transmission. Once the write operation returns to the application, the user can then access the buffer.

For kernel-mode data transmission, the kernel-mode process issues an **fp_rwuio** kernel service. A pointer to an **mbuf** structure is passed as a parameter. The data starting at a specified offset into the **mbuf** structure, for a length specified by the **m_len** field, is given to the Ethernet adapter for transmission.

Note: Once the kernel-mode process has issued the **fp_rwuio** call, it must not access the **mbuf** structure or structures again. If the process has requested a transmit acknowledgment, the process can access the **mbuf** structure once the transmit acknowledgment has occurred.

Depending upon the options specified with the **fp_rwuio** call, the device handler can call the **tx_fn** function. The **tx_fn** function is specified in the **entopen** entry point to notify the kernel-mode process of a completed transmission. Whether the device handler frees the **mbuf** structure depends upon the options specified in the **fp_rwuio** call.

Data Reception for the Ethernet Device Handler

Data reception for the Ethernet device handler depends on the type of process. When the address of a packet matches the address of one of the following:

- Address of the adapter specified in the device-dependent structure (DDS)
- Broadcast address
- Multicast address.

The adapter receives that packet and places it in the adapter receive buffer.

Note: The Version 3 Ethernet device handler does not support promiscuous addressing.

The adapter performs direct memory addressing of the data into the specified offset in the **mbuf** structure. This offset is calculated beforehand using the **data_off** field. Once the entire packet is in the mbuf the adapter interrupts the device handler. The device handler checks the **netid** and **type** field. If no match is found, the buffer is purged.

Data Reception in Kernel Mode

If the received data is for a kernel-mode caller, the device handler calls the **rx_fn** function that was specified at open time. The address of the **mbuf** structures that contain the kernel-mode process data is passed in the **rx_fn** function call. The process is responsible for freeing the **mbuf** structures that contain the received data.

Data Reception in User Mode

If the application has an outstanding **entselect** entry point for available data, the device handler calls the **selnotify** kernel service to notify the application that data is available.

If the application does not have an outstanding **entselect** entry point, the device handler queues up the data for the application upon receiving an **entselect** entry point.

Once the application has been notified of available receive data, the application issues a **read, readv, readx,** or **readvx** subroutine to get the incoming data. The device handler moves the data in the kernel space **mbuf** structure to the buffer specified by the application in the read call.

Return Values for the Ethernet Device Handler

The return codes for the users of the Ethernet device handler can be found in the **<sys/errno.h>** file. The return codes are defined as follows:

EACCES	Indicates that permission was denied because the device had not been initialized. Indicates that the Diagnostic mode open request was denied. Indicates that permission was denied because the call is from a kernel-mode process.
EAFNOSUPPORT	Indicates that the address family was not supported by protocol or that the multicast bit in address was not set.
EAGAIN	Indicates that the transmit queue is full or that the maximum number of opens had been reached.
EBUSY	Indicates that the open request was denied because the device was already open in Diagnostic mode or because the adapter was busy.
EEXIST	Indicates that the DDS already exists.
EFAULT	Indicates that an invalid address was specified.
EINTR	Indicates an interrupted system call.
EINVAL	Indicates an invalid range or opcode or that the Ethernet device handler was not in Diagnostic mode.
EIO	Indicates an I/O error.
EMSGSIZE	Indicates that the data returned was too big.
ENOBUFS	Indicates that no buffers were available.
ENOCONNECT	Indicates that no connection was established.
ENODEV	Indicates that no such device exists.
ENOENT	Indicates that there is no DDS.
ENOMEM	Indicates insufficient memory exists.
ENOMSG	Indicates that there was no message of desired type.
ENOSPC	Indicates that there was no space left on the device. (Multicast table is full.)
ENOTREADY	Indicates that the device was not ready. (First CIO_START operation not issued and not completed.)
ENXIO	Indicates that an attempt was made to use an unconfigured device.
EUNATCH	Indicates that the protocol driver was not attached.

Error Logging for the Ethernet Device Handler

The Ethernet device handler logs the following errors.

ERRID_ENT_ERR1	Indicates a permanent Ethernet adapter hardware error. This error can occur for any of the following reasons:

	• Mismatch between firmware version in microcode and ROS Level from the VPD. • First start command to the adapter failed to complete in a specified amount of time. • One or more of the fields in the VPD structure is invalid.
ERRID_ENT_ERR2	Indicates a temporary Ethernet adapter hardware error. This error can occur for any of the following reasons: • Adapter detected a parity error and reported this error to the device driver. • One of these transmit errors has occurred: Maximum Collisions, FIFO Underrun, Carrier Sense Lost, Clear To Send Lost, Transmit Time out, Packet Too Short or Too Large.
ERRID_ENT_ERR3	Indicates a permanent Ethernet adapter firmware error. • Adapter execute command returned with error bit set. • Adapter execute command (Report Configuration) returned invalid data.
ERRID_ENT_ERR5	Indicates an unknown system error caught by Ethernet device driver. • DMA kernel facilities failed. • DMA region facilities failed. • Receive buffer facilities (the **mbuf** structure) failed.
ERRID_ENT_ERR6	Indicates an ALERT for Transmit Carrier Detect Lost.

Device Dependent Structure for the Ethernet Device Handler

The Ethernet device driver provides a configuration method to the Ethernet adapter device handler. The ethernet configuration method allows for the changing of several Ethernet-specific parameters. Each of these parameters has a default value that should work for most applications. Each of these configuration parameters also has a specified range of valid values. The following are the configurable parameters for the Ethernet device driver:

Receive Data Transfer Offset

Indicates where in the receive buffer the packet data actually begins. Valid values range from 0 to 127. The default value of this field is 92.

Alternate Ethernet Address

Allows for changing the adapter unique address as it appears on the LAN network. This field is used in conjunction with the Enable Alternate Ethernet Address field. The address selected *must* be unique on the network. The default value is the hexidecimal value 02 60 8C 00 00 01.

Ethertype Field Offset

Indicates the offset in the packet where the Ethernet Type Field is located. Valid ranges are from 0 to 1513. The default value of this field is 12.

Transmit Queue Size

Indicates the number of transmit elements that can be queued up by the device handler. The default value of this field is 30.

Receive Queue Size

Indicates the number of receives that can be queued up by the device handler for each user space open. The default value for this field is 30.

Status Block Queue Size

Indicates the number of status elements that can be queued up by the device handler for each open user space. The default value of this field is 30.

Enable Alternate Ethernet Address

Indicates whether the unique Ethernet adapter address is the address that is supplied with (burned in) the adapter or an address that is supplied by the alternate ethernet address. Valid values are 0 and 1, where 0 indicates the user of the burned-in address and 1 indicates the user of the alternate ethernet address. The default value of this field is 0.

802.3 Ethertype Offset

Indicates the offset in the packet where the 802.3 ethertype field is located. Valid values range from 0 to 1513. The default value of this field is 14.

Adapter Connector Select

Indicates which one of the adapter's external connectors is being used. Valid values range from 0 to 1, where 0 indicates that the 15-pin DIX connector is selected and 1 indicates the BNC connector is selected. The default value of this field is 1.

Vital Product Data (VPD) Structure for the Ethernet Device Handler

The Vital Product Data is read from the Ethernet adapter by the POS registers. Although there is no specific order for entries, the **label_m**, **length_m**, and **data_m** fields are required and the following fields must always appear first:

- label
- length_msb
- length_lsb
- crc_msb
- crc_lsb.

The generalized structure for the VPD is as follows:

```
struct vpd  {
 uchar label[3];          /* ascii string for "VPD"      */
 ushort length;           /* length of 1/2 the structure */
 ushort  crc;             /* two bytes of CRC data       */
 uchar  label_1[3];       /* ascii string for VPD label  */
 uchar  length_1;         /* length of 1/2 this entry    */
 uchar data_1[X1-1];      /* ascii string of entry data  */
 .           .
 .           .
 .           .
 uchar label_n[3];         /* ascii string for VPD label  */
 uchar  length_n;          /* length of 1/2 this entry    */
 uchar  data_n[Xn-1];      /* ascii string of entry data  */
} dd_vpd;
```

The fields in this structure provide the following information:

label	Represents a three-byte ASCII string that always has the ASCII characters VPD.

length	Indicates the length of the structure from the beginning of the string indicated by the *label_1* parameter, to the end of the string indicated by the *data_n* parameter. This two-byte field represents the number of two-byte words.
crc	Represents the cyclic redundancy check (CRC) result that is performed on the entries starting at the string specified by the *label_1* parameter and progressing to the end of the string, specified by the *data_n* parameter. This is a two-byte field.
label_n	Describes the type of entry. This string starts with the ASCII character * (asterisk) character and ends with two predefined uppercase letters. This is a 3-byte ASCII string.
length_n	Indicates the number of two-byte words that this entry consumes. This is a one-byte field.
data_n	Represents the pertinent data for this entry. This is usually an ASCII string of characters of varying length. When specifying the network address, this field is hexadecimal.

Note: The Ethernet VPD structure is stored in the **<sys/ciouser.h>** file.

Device Characteristics Structure for the Ethernet Device Handler

The **ddi_cc_section_t** structure specifies the device characteristics structure for the Ethernet device handler and associated labels. This structure is defined in the **<sys/cioddi.h>** file and contains the following fields.

bus_type	Specifies the bus type. This field is used by the **i_init** kernel service.
bus_id	Specifies the bus ID. This field is used by BUSACC, **i_init** kernel service.
intr_level	Specifies the interrupt level. This field is used by the **i_init** kernel service.
intr_priority	Specifies the interrupt priority. This field is used by the **i_init** kernel service.
xmt_que_size	Specifies the size of the transmit queue for this adapter that is shared by all opens.
rec_que_size	Specifies the size of the receive queue for each open process.
sta_que_size	Specifies the size of the status queue for each open process.
rdto	Specifies the receive data transfer offset. This is offset in the **mbuf** receive data buffer where the data begins.

Ethernet Device Handler Hardware Characteristics Structure

The **ddi_ds_section_t** structure defines the device-dependent structure (DDS) for the Ethernet device handler. This structure is defined in the **<sys/entddi.h>** file and contains the following fields.

lname
Specifies the logical name of the device. This is an array of 4 ASCII characters that has the format of `entx`, where `x` is 0 to 3.

bus_mem_addr
Indicates the memory base address (MBA). Addresses are specified by the POS register setup. The following addresses are valid:

0x000C0000	0x000C4000	0x000C8000	0x000CC000
0x000D0000	0x000D4000	0x000D8000	0x000DC000

bus_mem_size
Specifies the memory base size (MBS). This field indicates the amount of bus-addressable RAM bytes contained on the adapter. This should normally be 16,384 (16K).

tcw_bus_mem_addr
Indicates the TCW bus memory address. This field points to the starting address of where DMA transfers can take place.

tcw_bus_mem_size
Indicates the TCW bus memory size. Points to the size of the address space that DMA transfers can take place in.

io_port Indicates the I/O base address. This field specifies addresses by the POS register setup. The following addresses are valid:

0x7280 0x7290 0x7680 0x7690
0x7A80 0x7A90 0x7E80 0x7E90

slot Indicates the I/O bus slot number. This value is required for accessing the POS registers for configuration of the Ethernet adapter.

dma_arbit_lvl Indicates DMA arbitration levels. Valid arbitration levels are 0 through 15. Level 15 always conflicts with the Micro Channel's host processor. This field accepts the following values:

0x0	Disabled
0x1	16–byte
0x2	32–byte
0x3	64–byte.

bnc_select Indicates transceiver selection. This field selects either the DIX connector or BNC connector. A value of 0 selects the DIX connector. A value of 1 selects the BNC connector.

use_alt_addr Indicates which network address to use. A value of 0 indicates a burned-in address. A value of 1 indicates an alternate address.

alt_addr Specifies the alternate network address. This is a 6-byte array.

type_field_off Specifies the type field offset. This field indicates the position inside the receive data packet where the compare for the type field (network ID) is to be performed.

net_id_offset Specifies the network ID offset. This field indicates the position inside the receive data packet where the compare for the IEEE 802.3 DSAP is to be performed.

Token-Ring Device Handler

The Token-Ring device handler is a component of the Communication I/O Subsystem. The Token-Ring device handler can handle one of the values specified by TOK_MAK_ADAPTERS Token-Ring adapters at a time.

The Token-Ring device handler interface consists of these 8 entry points:

tokconfig Provides functions for initializing, terminating, and querying the VPD of the Token-Ring device handler.

tokopen Initializes the Token-Ring device handler and allocates the required system resources.

tokclose Resets the Token-Ring device to a known state and frees system resources.

tokmpx Provides allocation and deallocation of a channel for the Token-Ring device handler.

tokwrite Provides the means for transmitting data.

tokread Provides the means for receiving of data.

tokselect Determines if a specified event has occurred on the Token-Ring device.

tokioctl Provides various functions for controlling the Token-Ring device handler. There are nine possible **tokioctl** operations:

IOCINFO	Supplies I/O character information.

CIO_START	Starts the device.
CIO_HALT	Halts the device.
CIO_QUERY	Queries device statistics.
CIO_GET_STAT	Gets device status.
TOK_GRP_ADDR	Sets the group address.
TOK_FUNC_ADDR	Sets functional addresses.
TOK_QVPD	Queries VPD.
TOK_RING_INFO	Queries Token-Ring Information.

Network Recovery Mode for the Token-Ring Device Handler

When the Token-Ring device handler detects a hardware failure (for example, a lobe wire fault), the device handler enters Network Recovery mode. In Network Recovery mode, the device handler recycles the Token-Ring adapter while trying to recover from failure. The Token-Ring device handler continues the recovery logic until one of these three conditions occur:

- The device handler succeeds.
- An unrecoverable error is detected.
- The last **tokclose** entry point is received.

The user decides when to give up his recovery attempt.

When the Token-Ring device handler enters Network Recovery mode, the device handler notifies the user with a CIO_ASYNC_STATUS status block.

While in Network Recovery mode, the Token-Ring device handler is not fully functional. Operations that initiate activity on the Token-Ring device should not be issued, in particular, the **tokwrite** entry point.

All other Token-Ring commands are acceptable.

When the Token-Ring device handler leaves Network Recovery mode, the device handler notifies the user with a CIO_ASYNC_STATUS status block.

Note: The network recovery scheme assumes that the upper layer protocols handle the functions of the data link layer. These functions are defined in the ISO Open Systems Interconnection model. For example, the transmission retries once the network error is cleared.

Data Transmission for the Token-Ring Device Handler

The Token-Ring device handler supports user- and kernel-mode transmission.

Kernel-Mode Data Transmission

For data transmission, the kernel-mode process issues an **fp_rwuio** kernel service. A pointer to an **mbuf** structure is passed as a parameter. The data starting at an offset of the **m_off** field into the **mbuf** structure, for the length of the **m_len** field, is given to the Token-Ring adapter for transmission.

Note: Once the kernel-mode process has issued the **fp_rwuio** kernel service, the process must not access the **mbuf** structure again. If the kernel-mode process has requested a transmit acknowledgment, the process can access the **mbuf** structure once the transmit acknowledgment has occurred.

Depending on which options the **fp_rwuio** kernel service specified, the device handler can kernel service the **tx_fn** function. This function was specified in the **tokopen** entry point to notify the kernel-mode process of transmit complete. The device handler's freeing of the **mbuf** structure depends on the options specified in the **fp_rwuio** kernel service.

User-Mode Data Transmission

A user-mode caller issues a **write, writev**, **writex**, or **writevx** subroutine and then specifies a buffer address. The Token-Ring device handler copies the buffer to a kernel-space **mbuf** structure. The device handler then gives the Token-Ring adapter the packet for transmission. Once the **tokwrite** entry point returns, the user can then access the buffer.

Data Reception for the Token-Ring Device Handler

The Token-Ring adapter receives a packet that contains one or more of these four items:

- Address of the adapter (or address specified in the DDS)
- Broadcast address
- Group address
- Functional address.

The packet is placed in an adapter receive buffer.

Note: The AIX Token-Ring device handler does not support promiscuous addressing.

When a packet is placed in the device handler's receive buffer, an interrupt is generated. The packet is moved to an offset in the receive buffer as specified by the Receive Data Transfer Offset (RDTO) value contained in the DDS. The device handler checks the netid field (DSAP) for a match in the Network ID table. If the device handler does not find a match, the packet is purged from the device handler's receive buffer.

Kernel-Mode Data Reception

If the received data is for a kernel-mode process, the device handler calls the **rx_fn** function that was specified at open time. The address of the **mbuf** structures that contain the kernel-mode process's data is passed in the **rx_fn** function call. The kernel-mode process is responsible for freeing the **mbuf** structures that contain the received data.

User-Mode Data Reception

If a user-mode application has an outstanding **tokselect** entry point for data available, the device handler calls the **selnotify** kernel service to notify the application that data is available.

If the application does not have an outstanding **tokselect** entry point, the device handler queues up the data for the application upon receiving a **tokselect** entry point.

Once the application has been notified that receive data is available, the application issues a **read**, **readv**, **readx**, or **readvx** subroutine to retrieve the incoming data. The device handler moves the data in the kernel space **mbuf** structure to the buffer specified by the application in the read call.

Token-Ring Operation Results

Return codes for users of the Token-Ring device handler can be found in the **<sys/tokuser.h>** file. The return codes are defined as follows:

TOK_ADAP_CONFIG	Adapter configuration failed.
TOK_ADAP_INIT_PARMS_FAIL	Adapter failed to take the adapter initialization parameters.
TOK_ADAP_INIT_FAIL	Adapter initialization failed.
TOK_ADAP_INIT_TIMEOUT	Adapter initialization timed out.
TOK_LOBE_MEDIA_TST_FAIL	The Token-Ring adapter's Lobe Media test failed.
TOK_PHYS_INSERT	Unable to insert on network.
TOK_ADDR_VERIFY_FAIL	Address verification failed.
TOK_REQ_PARMS	Error in the request parameters sequence.

TOK_LOBE_WIRE_FAULT	The adapter has detected an open or short circuit in the lobe data path.
TOK_AUTO_REMOVE	Adapter has detected an internal hardware error.
TOK_REMOVED_RECEIVED	Remove adapter command from the LAN manager was received.
TOK_SIGNAL_LOSS	Adapter detected an absence of a receive signal.
TOK_RING_STATUS	The adapter returned a ring status indicating a Token-Ring error condition.
TOK_ADAP_CHECK	An adapter check has occurred.
TOK_CMD_FAIL	Adapter command failure.
TOK_TX_ERROR	An error occurred during transmission of the packet.
TOK_PIO_FAIL	Program I/O failure.
TOK_RCVRY_THRESH	Network Recovery mode entry threshold has been exceeded.
TOK_NO_GROUP	Group address is already specified. Unable to set group address.
TOK_NO_PARMS	Device already started. Unable to set the adapter parameters.
TOK_NO_RING_INFO	There is no Token-Ring information currently available.
TOK_RING_BEACONING	The Token-Ring is experiencing a beacon condition.
TOK_RING_POLL	The Token-Ring adapter's ring poll test failed.
TOK_RING_RECOVERED	The Token-Ring has recovered from the beaconing condition.
TOK_LOADER_FAIL	The download of the loader program failed.
TOK_UCODE_FAIL	The download of the microcode program failed.
TOK_MC_ERROR	A Micro Channel error was detected by the Token-Ring device handler.

Error Logging for the Token-Ring Device Handler

The AIX Token-Ring device handler logs the following errors:

ERRID_TOK_WRAP_TST
Indicates that wrap test failed in the insertion process.

ERRID_TOK_BEACON1
Indicates beaconing during the insertion process.

ERRID_TOK_DUP_ADDR
Indicates a duplicate station address.

ERRID_TOK_RMV_ADAP1
Indicates that a Remove Adapter command was received during the insertion process.

ERRID_TOK_ERR5
Indicates that an unknown adapter hardware error occurred during the insertion process.

ERRID_TOK_WIRE_FAULT
Indicates a wire-fault condition on the ring.

ERRID_TOK_AUTO_RMV
Indicates an adapter autoremove error.

ERRID_TOK_RMV_ADAP2
Remove adapter command from the LAN manager.

ERRID_TOK_BEACON2
Indicates ring beaconing.

ERRID_TOK_ERR10
Indicates that the ring was in a beaconing condition for a time shorter than the hard-error detection timer.

ERRID_TOK_BEACON3
Indicates that ring beaconing occurred for less than 52 seconds.

ERRID_TOK_ESERR
Indicates excessive soft errors for the ring.

ERRID_TOK_CONGEST
Indicates that an adapter on the ring is experiencing excessive congestion.

ERRID_TOK_ADAP_CHK
Indicates that a Token-Ring adapter check has occurred.

ERRID_TOK_NOMBUFS
Indicates that a Token-Ring device handler request for an **mbuf** structure was denied.

ERRID_TOK_DOWNLOAD
Indicates that the microcode download failed.

ERRID_TOK_BAD_ASW
Indicates an incompatible microcode and adapter.

ERRID_TOK_ERR15
Indicates that the Token-Ring device handler caught an unknown system error.

ERRID_TOK_RCVRY_ENTER
Indicates that the Token-Ring device handler has entered Network Recovery mode.

ERRID_TOK_RCVRY_EXIT
Indicates that the Token-Ring device handler has exited Network Recovery mode.

ERRID_TOK_RCVRY_TERM
Indicates that the Token-Ring device handler has terminated Network Recovery mode.

ERRID_TOK_MC_ERR
Indicates that the Token-Ring device handler has detected a Micro Channel error.

ERRID_TOK_TX_ERR
Indicates that the Token-Ring device handler has detected a transmission error.

ERRID_TOK_PIO_ERR
Indicates that the Token-Ring device handler has detected a program I/O error.

The error types listed above are defined in the **<sys/errids.h>** include file.

Multiprotocol (MPQP) Device Handler Interface

The Multiprotocol (MPQP) device handler is a component of the Communication I/O Subsystem. The MPQP device handler interface is made up of the following seven entry points:

mpclose	Resets the 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.
mpconfig	Provides functions for initializing and terminating the MPQP device handler and adapter.
mpioctl	Provides various functions for controlling the MPQP device.

	CIO_START	Initiates a session with the MPQP device handler.
	CIO_HALT	Ends a session with the MPQP device handler.
	CIO_QUERY	Reads the counter values accumulated by the MPQP device handler.
	CIO_GET_STATUS	Gets the status of the current MPQP adapter and device handler.
	MP_START_AR	Puts the MPQP port into Autoresponse mode.
	MP_STOP_AR	Permits the MPQP port to exit Autoresponse mode.
	MP_CHG_PARMS	Permits the DLC to change certain profile parameters after the MPQP device has been started.
mpopen		Opens a channel on the MPQP device for transmitting and receiving data.
mpmpx		Provides allocation and deallocation of a channel.
mpread		Provides the means for receiving data to the MPQP device.
mpselect		Provides the means for determining which specified events have occurred on the MPQP device.
mpwrite		Provides the means for transmitting data to the MPQP device.

Binary Synchronous Communication (BSC) with the MPQP Adapter

The Multiprotocol Quad Port (MPQP) adapter software performs low level BSC frame type determination to facilitate character parsing at the kernel-mode process level. Frames that are 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. Buffer overrun errors that are not logged are an exception.

Note: In BSC communcations, the caller receives either a message type or an error status.

Read operations must be performed using the **readx** subroutine since the **read_extension** structure is needed to return BSC function results.

BSC Message Types Detected by the MPQP Adapter

BSC message types are defined in the **<sys/mpqp.h>** file. The MPQP adapter can detect the following message types:

- MP_ACK0
- MP_ACK1
- MP_WACK
- MP_NAK
- MP_ENQ
- MP_EOT
- MP_RVI
- MP_DISC
- MP_SOH_ITB
- MP_SOH_ETB
- MP_SOH_ETX
- MP_SOH_ENQ
- MP_STX_ITB

- MP_STX_ETB
- MP_STX_ETX
- MP_STX_ENQ
- MP_DATA_ACK0
- MP_DATA_ACK1
- MP_DATA_NAK
- MP_DATA_ENQ.

BSC Receive Errors Logged by the MPQP Adapter

The 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 MPQP adapter:

- Receive overrun because the card did not keep up with line data.
- Driver did not supply buffer in time for data.
- A CRC or LRC framing error.
- Parity error.
- CTS time out while the adapter is in Autoresponse mode.
- 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 and 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** value is returned.

Error Logging for the Multiprotocol (MPQP) Device Handler

The large percent of errors logged by the AIX Multiprotocol (MPQP) device handler detect problems in the configuration or network equipment rather than software defects. The following errors are logged for the MPQP device handler:

DSR On Timeout

The DSR failed to come on. This error is a result of the modem failing to signal a ready condition. The following action is recommended:

- Ensure that the configuration profiles match those of the actual modem configuration.
- Check and make sure the following are true:
  - The cable is connected to the correct port on the fan-out box (FOB).
  - The modem is powered on.
  - The FOB connection on the back of the card is correct.
  - The correct cable is connected in the correct FOB.
- If running switched line configurations, ensure that the modem is connecting to the line, the correct phone number is being used, and that the command stream is correct (if using autodial modems).
- If running with null modems or cross-over cables, ensure that the data set read (DSR) is wrapped to DTR and clear to send (CTS) is wrapped to request to send (RTS).

DSR Off Timeout

When running on a switched line, this error is caused by starting a call while the DSR is on. If the DSR is on before the adapter drives DTR, the adapter assumes the previous call is still in progress.

Note: This is a security measure to ensure the previous session was terminated before the next call starts.

The following action is recommended:

- Ensure that the line is a switched line setup. If it is leased, check the profile parameters.
- Check the modem setup. Ensure the modem specification to determine whether DSR is driven high all the time. If so, this modem is not compatible for switched line use with the MPQP adapter using the AIX provided software.

CTS Dropped on Transmit

The modem is dropping a CTS before the frame has been completely transmitted. The following action is recommended:

- Check the modem to ensure it is operating properly.
- If not running a multidrop line, the continuous carrier option causes an RTS to be driven continuously and may prevent the modem from dropping CTS.

DSR Dropped

The link went down during the session. This is a fatal error for the MPQP adapter. The application must re-initiate the session. The following action is recommended:

- Use the modem specification to determine why the modem is dropping DSR.
- Check the connection.

Receive Data Error

If the data is not received correctly, the driver logs an error and the application can recover by retransmitting. A certain amount of receive errors may be normal and is not cause for concern unless they errors become excessive and significantly degrade throughput. The following action is recommended:

- Ensure that the local and remote modems are compatible and are configured in the same way.
- Ensure that the local and remote computers are configured similarly. For example, if one station is sending SDLC NRZI data, the other machine must be configured to send SDLC NRZI data also.
- Check the quality of the phone lines or connecting hardware. An excessive amount of receive errors can be a sign of line quality problems.
- Ensure that the modem is running within specified range of valid baud rates.
- Check the clocking options in the profile and the modem. If both the card and the modem try to provide the data clocks, data is received incorrectly.

Adapter Not Present or Not Functioning, Adapter IPL Timeout

The power-on-self test failed or the driver was unable to load the adapter software. The following action is recommended:

- Make sure the board is seated properly in the slot.
- Check the adapter diagnostics.
- Check the integrity of the adapter software file.

DMA Buffer Not Allocated

This error is usually caused by a resource outage in the system. The following action is recommended:

- Increase the number of buffers in the pool.
- Reduce unnecessary load on the machine. For example, disable unused tty ports or reduce simultaneous disk activity.
- Check to see if the amount of main memory is adequate.

Transmit Underrun, Receive Overrun

This error indicates the card is not keeping up with the data rate. This error should not occur with the MPQP adapter as the data is buffered on the card. The following action is recommended:

- Check the quality of the line.
- Verify that the modem and the system are not both trying to provide data clocks.

Transmit Failsafe Timeout Expired

Transmit timeouts usually occur for one of two reasons:

- The port is not detecting the transmit clock. The port is configured for modem clocking and no clock is provided is by the modem. The port is configured for DTE clocking and both the port and the modem are providing clocks.
- The port does not detect CTS on by the modem.

The following action is recommended:

- Verify port configuration profiles and setup.
- Ensure that the modem equipment and cables are functional.

X.21 Timeout

This error applies to X.21 protocol users only and is caused by a timeout while trying to establish a connection. This error can be caused by the X.21 failing to recognize the MPQP state change. The following action is recommended:

- Ensure that the DCE is in service on both the modem and the network.

X.21 Unexpected Clear During Call Establishment

This error applies to X.21 protocol users only and is caused by the X.21 DCE asking the MPQP adapter to terminate. The following action is recommended:

- Consult the X.21 specification form more information about the error.

X.21 Unexpected Clear During Data Phase

This error applies to X.21 protocol users only and indicates that after the call completed successfully data phase was entered and the call cleared. This may indicate a normal termination initiated by the network. The following action is recommended:

- Determine whether a session was aborted or whether the clear was normal.
- Consult the X.21 specification for more information about the error.

X.21 Call Progress Signal

This error applies to X.21 protocol users only and indicates the call did not complete or additional network information was received. The following action is recommended:

- Using the call progress signal value determine if an error scenario occurred.
- Consult the X.21 specification for more information about the error.

Description of the Multiprotocol (MPQP) Card

The MPQP card is a four-port multiprotocol adapter that supports BSC and SDLC on the EIA232–D, EIA422–A, X.21, and V.35 physical interfaces. The MPQP card uses the microchannel bus and communicates with the adapter with programmed I/O (PIO) and first party DMA (bus master).

The adapter has 512K bytes of RAM and an Intel 80C186 processor. There are 16 dedicated DMA channels between the RAM and the physical ports. The drivers and receivers for each of the electrical interfaces reside on a daughter board that is joined to the base card with two 60-pin connectors.

A shielded cable attaches to the 78-pin D-shell connector on the daughter board and routes all signals to a fan-out box. The fan-out box has nine standard connectors that support each possible configuration on each port. Standard 15-pin or 25-pin cables are used between the fanout box and the modem for each electrical interface.

The following are the interfaces available on each port:

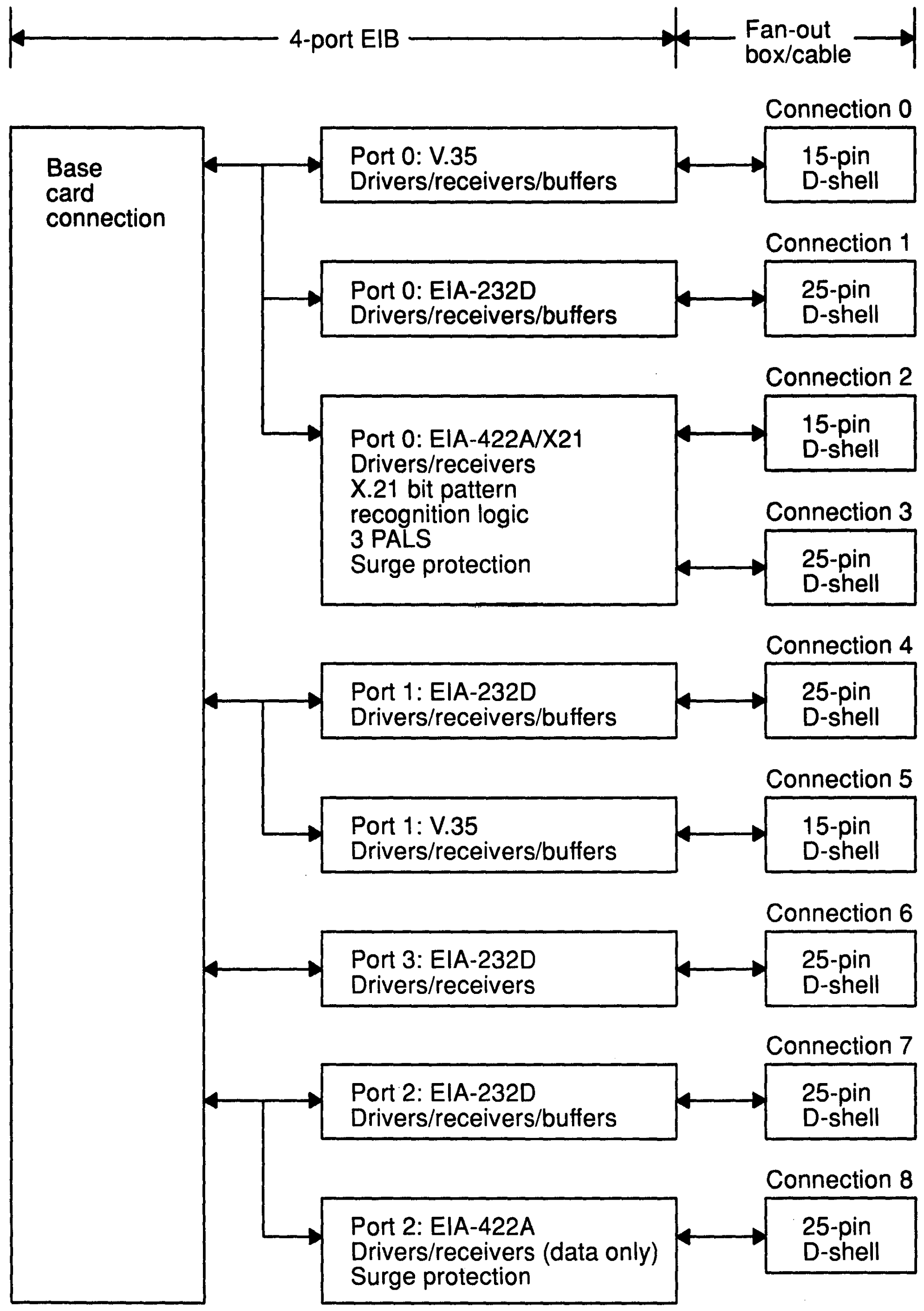
Port Configurations				
Number	**Port-0**	**Port-1**	**Port-2**	**Port-3**
1	EIA-232D	EIA-232D	EIA-232D	EIA-232D
2	EIA-422A	EIA-232D	EIA-232D	EIA-232D
3	V.35 EIA-232D	EIA-232D V.35	EIA-232D EIA-232D	EIA-232D EIA-232D
4	X.21	EIA-232D	EIA-232D	EIA-232D
5	EIA-422A	V.35	EIA-232D	EIA-232D
6	V.35	V.35	EIA-232D	EIA-232D
7	X.21	V.35	EIA-232D	EIA-232D
8	EIA-232D	EIA-232D	EIA-422A	EIA-232D
9	EIA-422A	EIA-232D	EIA-422A	EIA-232D
10	V.35 EIA-232D	EIA-232D V.35	EIA-422A EIA-422A	EIA-232D EIA-232D
11	X.21	EIA-232D	EIA-422A	EIA-232D
12	EIA-422A	V.35	EIA-422A	EIA-232D
13	V.35	V.35	EIA-422A	EIA-232D
14	X.21	V.35	EIA-422A	EIA-232D

Port 0	EIA232–D, EIA422–A, X.21, and V.35. This port has the highest DMA priority. The EIA–422A interface on this port has data and clock signals.
Port 1	EIA232–D and V.35.
Port 2	EIA232–D and EIA422–A (data only). The EIA–422A interface on Port 2 only has data signals.
Port 3	EIA232–D. This port has the lowest priority.

* Adheres to CCITT X.21 dial specifications.

The following depicts the mapping of physical interfaces to the fan-out box connectors:

Block Diagram


4-port EIB
Fan-out box/cable
Base card connection
Port 0: V.35 Drivers/receivers/buffers
Port 0: EIA-232D Drivers/receivers/buffers
Port 0: EIA-422A/X21 Drivers/receivers X.21 bit pattern recognition logic 3 PALS Surge protection
Port 1: EIA-232D Drivers/receivers/buffers
Port 1: V.35 Drivers/receivers/buffers
Port 3: EIA-232D Drivers/receivers
Port 2: EIA-232D Drivers/receivers/buffers
Port 2: EIA-422A Drivers/receivers (data only) Surge protection
Connection 0 15-pin D-shell
Connection 1 25-pin D-shell
Connection 2 15-pin D-shell
Connection 3 25-pin D-shell
Connection 4 25-pin D-shell
Connection 5 15-pin D-shell
Connection 6 25-pin D-shell
Connection 7 25-pin D-shell
Connection 8 25-pin D-shell

The following modem interfaces are supported by each physical interface:

	Call Establishment Protocol		
Physical Interface	**Leased**	**Manual Switched**	**Autodial**
EIA232-D	X	X	X
EIA422-A	X		
V.35	X		
X.21	X		X*

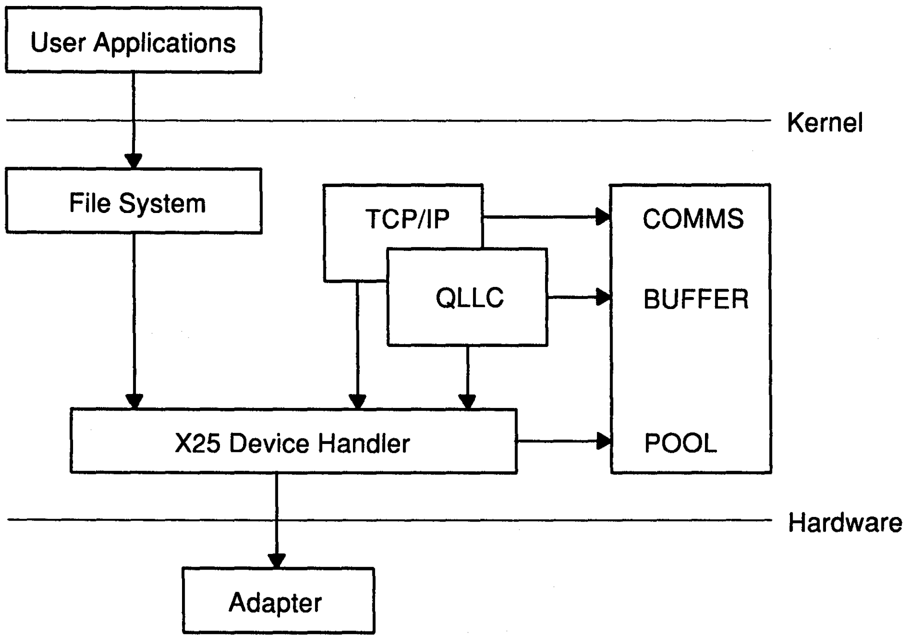
X.25 Device Handler

The AIX operating system uses special files to refer to specific hardware devices and device drivers. Special files are used like other files. They have path names that appear in a directory and access protection like ordinary files.

When a user program requests I/O using system calls, control is transferred to the kernel. If the call is to a special file, the kernel transfers control to the corresponding device driver. Similarly, when the IBM X.25 Interface Co-Processor/2 signals an interrupt, the kernel passes the interrupt to the X.25 device handler's interrupt handler, which processes it and returns to the kernel.

The X.25 device handler must conform to these system interfaces by offering a standard set of entry points. Entry points are called by the AIX kernel, not directly by the user program.

How the X.25 Device Handler Fits into the AIX System



The X.25 device handler, **/dev/x25s***n* is a multiplexed device driver. A multiplexed device driver permits multiple **open** subroutines. The minor device number, *n*, specifies which port or adapter is referred to. In addition, the port supports multiple X.25 connections on each multiplexed channel. Each X.25 connection is associated with one CIO_START operation.

The following is a list of the X.25 device handler entry points:

x25sopen	Initializes a channel into the X.25 device handler.
x25smpx	Provides the means to allocate and deallocate a channel into the X.25 device handler.
x25sclose	Closes an X.25 device handler channel.
x25sselect	Determines if a specified event occurred on a device.
x25sread	Provides the means to receive data from the X.25 adapter.
x25swrite	Provides the means to send data to the X.25 adapter.
x25sioctl	Provides various functions for controlling the X.25 device.

The following is a list of the X.25 ioctl operations:

IOCINFO	Identifies a device.
CIO_DNLD	Downloads a task.
CIO_START	Starts a session.
CIO_HALT	Halts a session.
CIO_QUERY	Queries a device.
CIO_GET_STAT	Gets device statistics.
X25_REJECT	Rejects a call.
X25_QUERY_SESSION	Queries a session.
X25_ADD_ROUTER_ID	Adds a router ID.
X25_DELETE_ROUTER_ID	Deletes a router ID.
X25_QUERY_ROUTER_ID	Queries a router ID.
X25_LINK_CONNECT	Connects a link.
X25_LINK_DISCONNECT	Disconnects a link.
X25_LINK_STATUS	Returns the status of the link.
X25_LOCAL_BUSY	Enables or disables receiving of data packets on a port.
X25_COUNTER_GET	Gets a counter
.X25_COUNTER_WAIT	Waits for the contents of a counter to change.
X25_COUNTER_READ	Reads the contents of a counter.
X25_COUNTER_REMOVE	Removes a counter from the system.
X25_DIAG_IO_WRITE	Writes to an I/O register on the IBM X.25 Interface Co-Processor/2.
X25_DIAG_IO_READ	Reads to an I/O register on the IBM X.25 Interface Co-Processor/2.
X25_DIAG_MEM_WRITE	Writes memory to the IBM X.25 Interface Co-Processor/2 from a user's buffer.
X25_DIAG_MEM_READ	Reads memory from the IBM X.25 Interface Co-Processor/2 into a user's buffer.
X25_DIAG_TASK	Provides the means to download the diagnostics task on to the card.

X.25 Programming Interfaces

How the interface between the X.25 device handler and the AIX kernel is presented depends on the process using the device handler. Two types of processes use the device handler, user-mode processes and kernel-mode processes. In both cases, the kernel itself does some processing before passing the call to the device handler.

For example, consider the **x25sioctl** entry point. A user-mode process sees this call in one of two ways:

- **int ioctl (***fildes, cmd, arg***)**
- **int ioctlx (***fildes, cmd, arg, ext***)**

The *fildes* parameter is a pointer to a file descriptor returned by the **open** subroutine. A return code of –1 indicates an unsuccessful operation, a 0 (zero) value indicates success. If –1 is returned, the kernel sets the **errno** global variable.

A kernel-mode process sees the **x25sioctl** entry point in the following manner:

int fp_ioctl (*fp, cmd, arg, ext***)**

The *fp* parameter is a pointer to a **file** structure, set on return from the **fp_open** kernel subroutine. The return value is either 0 (OK) or the error code itself.

Whether a process is a user mode or kernel-mode process, the X.25 device handler entry point that the kernel actually called is:

int x25sioctl (*devno, cmd, arg, devflag, chan, ext***)**

where

cmd, arg	Specify the values passed by the user-mode process or kernel-mode process.
ext	Specifies the value passed to the **ioctlx** subroutine or **fp_ioctl** kernel service. This parameter is NULL if the **ioctl** subroutine was called.
devno, chan	Identifies the major and minor device numbers and multiplexed channel number. These values are mapped in the kernel to < *pid, fildes* > for a user-mode process, or < *pid, fp* > for a kernel-mode process.
devflag	Identifies a flag word supplied by the kernel. This word indicates how the channel was opened, and whether the call was from a user-mode process or a kernel-mode process.

The **x25sioctl** entry point returns either 0 (OK), or –1 (error). If there is an error, it places the error code in **u.u_error** in the calling process's user block.

Note: The kernel can reject a system or subroutine call before it ever reaches the device handler. In that case, the kernel can return values of the **errno** global values other than those listed in this document.

The other entry points can be mapped to system calls or kernel subroutine calls in a similar fashion. A noteworthy special case is that of the **x25smpx** entry point. There is no **mpx** or **devmpx** call available directly to the calling process. The **x25smpx** entry point is called by the kernel in response to either of the following:

- An **open** subroutine or **fp_open** kernel service *before* calling the **x25sopen** entry point
- A **close** subroutine or **fp_close** kernel service *after* calling the **x25sclose** entry point.

X.25 Device Handler Modes

Four modes are provided for running the X.25 device handler. These modes are indicated by extensions added to the special device name on the **x25sopen** entry point. The following are the three available extensions:

/dev/x25s*n* Starts the device handler on the next available port.

/dev/x25s*n***/D** Opens the device handler in Diagnostic mode. The X.25 device handler provides a set of **ioctl** operations for running diagnostics on the adapter. These operations can be used by any program that opens the **/dev/x25s***n***/D** special file. Opening this special file is restricted as follows:

- Only one process per minor device number can open the file at a time.
- The file cannot be opened while there are any other open channels on that minor device number (as the microcode on the adapter must be changed to run diagnostics). The diagnostics cannot coexist with the X.25 microcode.
- Only processes that have appropriate permission can open the file.

Note: It is not possible to open a session on a channel opened in Diagnostic mode. It follows, opening the device in diagnostic mode does not require that a user start a session with the CIO_START operation.

/dev/x25s*n***/M** Opens the device handler for reading and writing data to the monitor facilities on the IBM X.25 Interface Co-Processor/2. The **/dev/x25s***n***/M** special file can be used to read and write data to the monitor facilities of the adapter. Opening this special file is restricted as follows:

- Only one process per minor device number at a time can open the file.
  - Only processes that have appropriate permission can open the file.

/dev/x25s*n***/R** Opens the device handler for updating the routing table. The X.25 device handler routes incoming call packets to listening applications according to its current routing table. The routing table can be managed by any program that opens the **/dev/x25s***n***/R** special file. Access to this special file is restricted as follows:

- Only one process at a time can open the file (the minor device number is ignored).
- Only processes that have both NET_CONFIG and appropriate permission can open the file.

  The routing process can use **x25sioctl** operations to add or delete routing table entries, and to query the ID of the process listening for a specified routing table entry. Other processes can listen for incoming calls by specifying which entry in the routing table they are listening for.

  Note: Although a routing table entry is not restricted to one minor device number, an individual listen request must be issued for each minor device on which a user wishes to listen for that specification.

While the **/dev/x25s***n***/R** file is open, other users are not permitted to start listening for incoming calls (using any minor device number). Any CIO_START operation with a session of type SESSION_SVC_LISTEN is rejected.

Sessions with the X.25 Device Handler

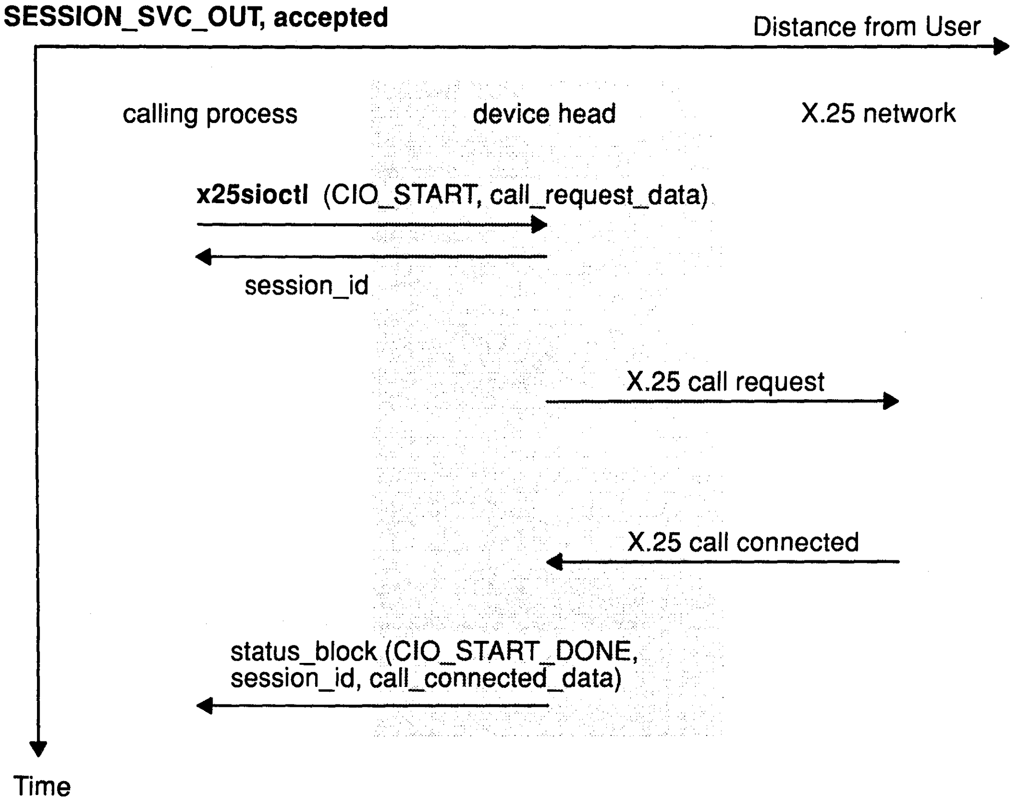
A session with the X.25 device handler lasts from a CIO_START operation to a CIO_HALT operation. A session must be established before a process can issue calls to the **x25sread** or **x25swrite** entry points. A session can be used for several purposes, identified by the associated session types:

SESSION_PVC	The session is used for a permanent virtual circuit.
SESSION_SVC_OUT	The session is used for a switched virtual circuit (SVC) initiated by the local DTE.
SESSION_SVC_IN	The session is used for a SVC initiated by the remote DTE.
SESSION_SVC_LISTEN	The session is used to receive incoming calls from remote DTEs. When an incoming call is received, it can either be rejected on this session or a new session (of type SESSION_SVC_IN) must be started to accept the call. The call is then associated with the new session, and the listening session continues to receive incoming calls.
SESSION_MONITOR	The session is used to receive and transmit packet monitor information.

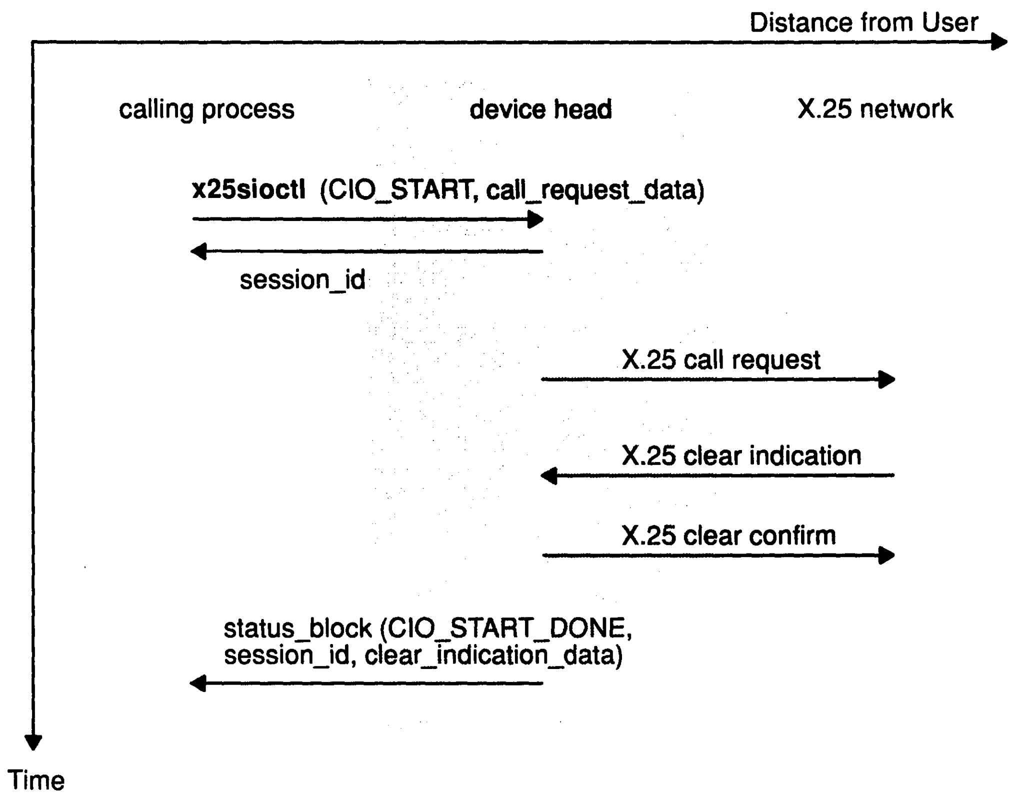
When establishing an X.25 session the following sequences may occur:

- A session of type SESSION_SVC_OUT is started and accepted.
- A session of type SESSION_SVC_OUT is started and rejected.
- A session of type SESSION_SVC_ LISTEN is started.
- A session of type SESSION_SVC_IN is started to accept the incoming call or an X25_REJECT operation rejects the call on the session.
- A session of type SESSION_PVC is started for a permanent virtual circuit.
- A session of type SESSION_MONITOR is started to receive and transmit packet monitor information.

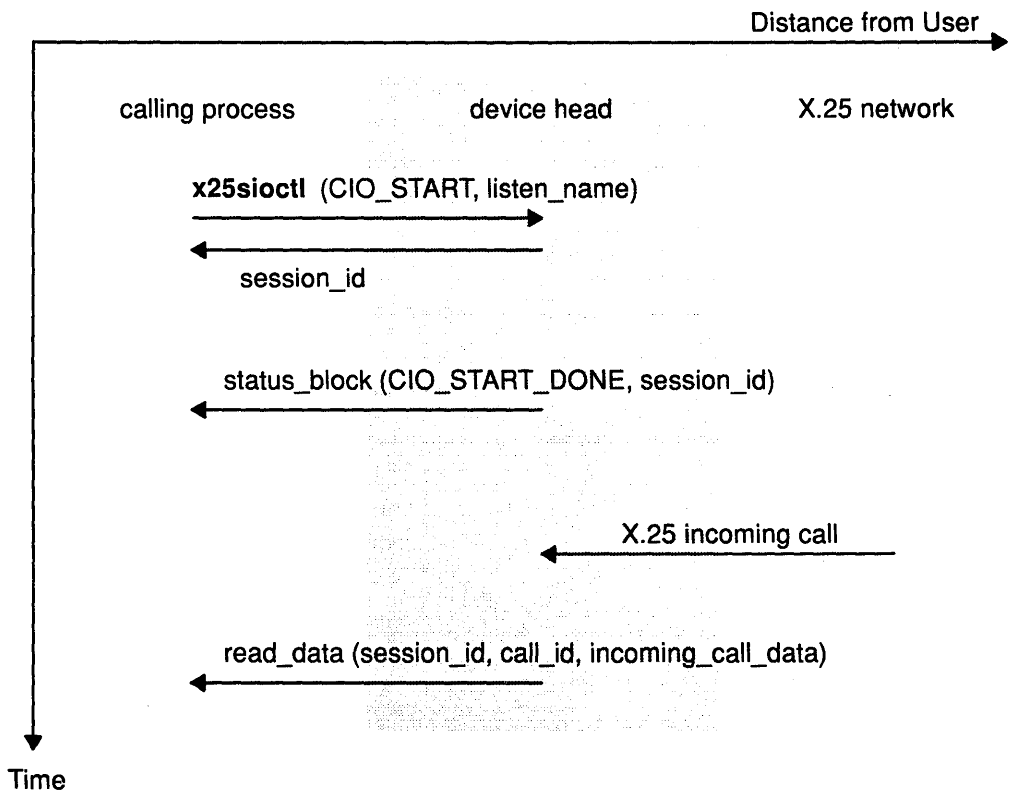
SESSION_SVC_OUT, accepted



SESSION_SVC_OUT, rejected

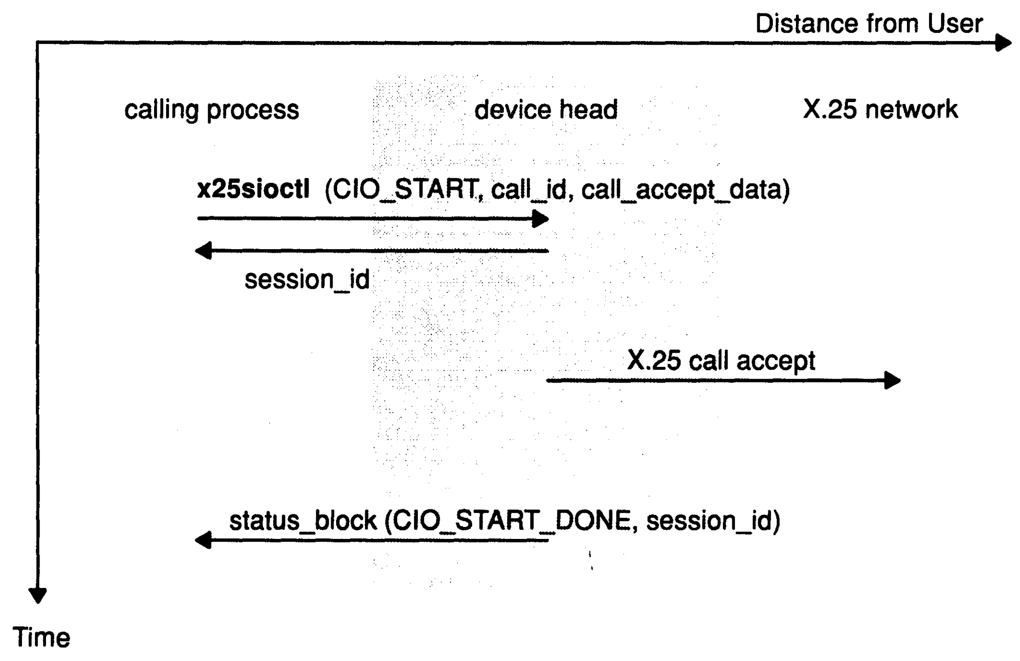


SESSION_SVC_LISTEN

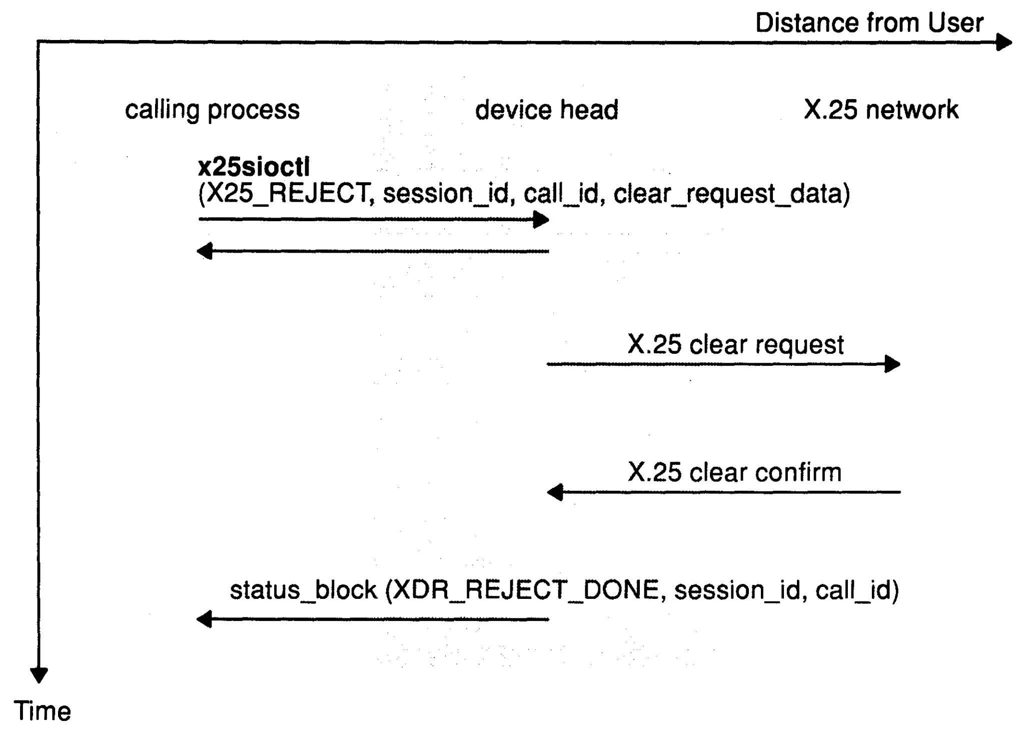


Distance from User
calling process
device head
X.25 network
x25sioctl (CIO_START, listen_name)
session_id
status_block (CIO_START_DONE, session_id)
X.25 incoming call
read_data (session_id, call_id, incoming_call_data)
Time

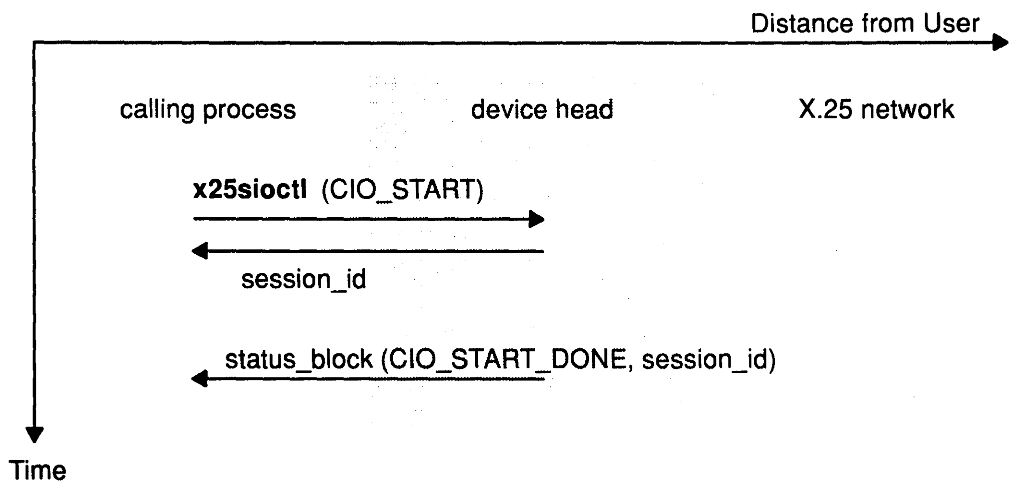
SESSION_SVC_IN


Distance from User
calling process
device head
X.25 network
x25sioctl (CIO_START, call_id, call_accept_data)
session_id
X.25 call accept
status_block (CIO_START_DONE, session_id)
Time

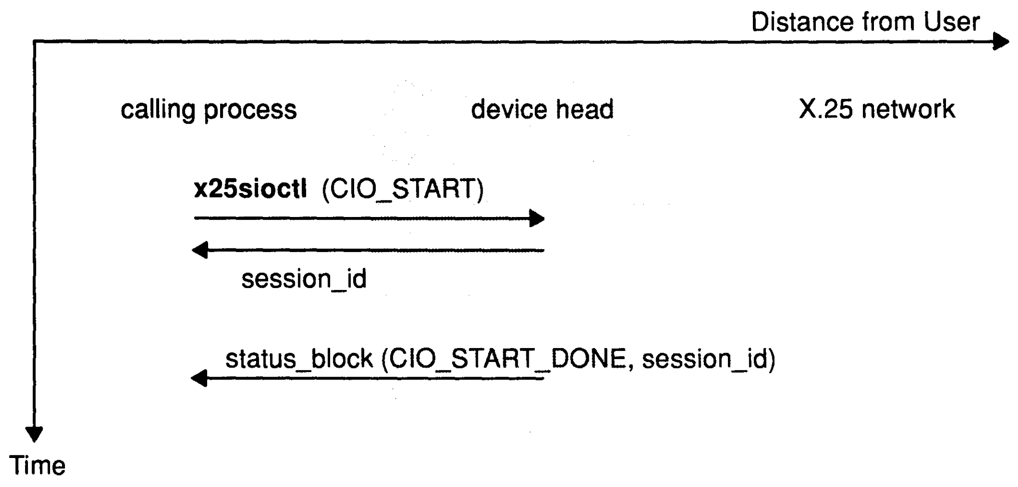
X25_REJECT



SESSION_PVC



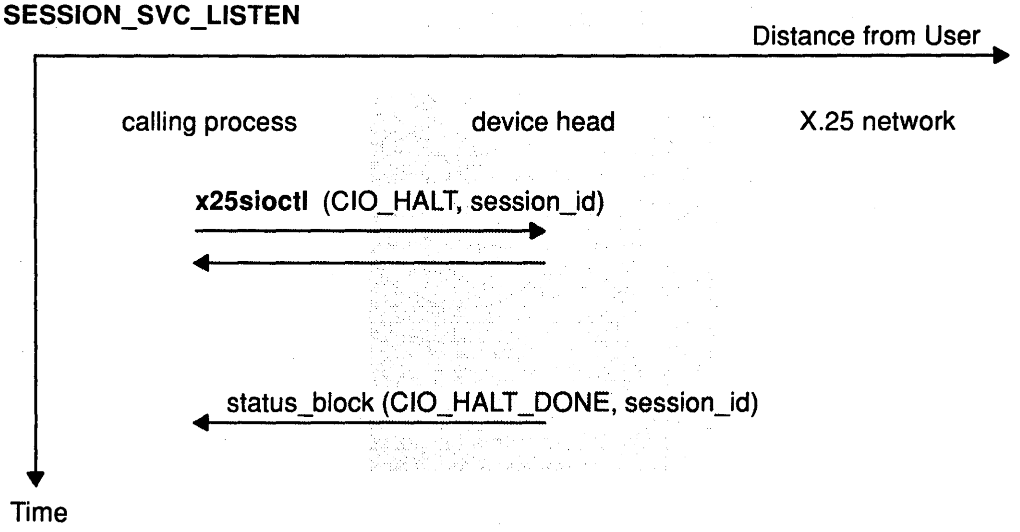
SESSION_MONITOR



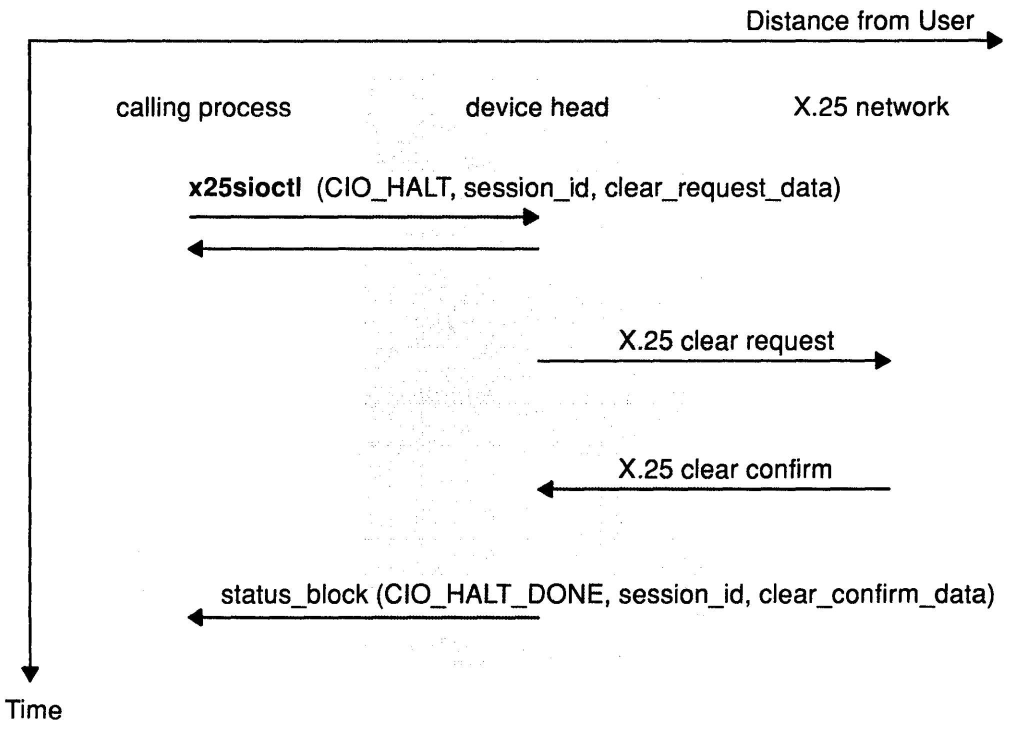
When terminating an X.25 session the following sequences may occur:

- A session of type SESSION_SVC_ LISTEN is halted.
- The session is cleared locally.
- The session is cleared remotely.
- A session of type SESSION_PVC is halted for a permanent virtual circuit.
- A session of type SESSION_MONITOR is halted.

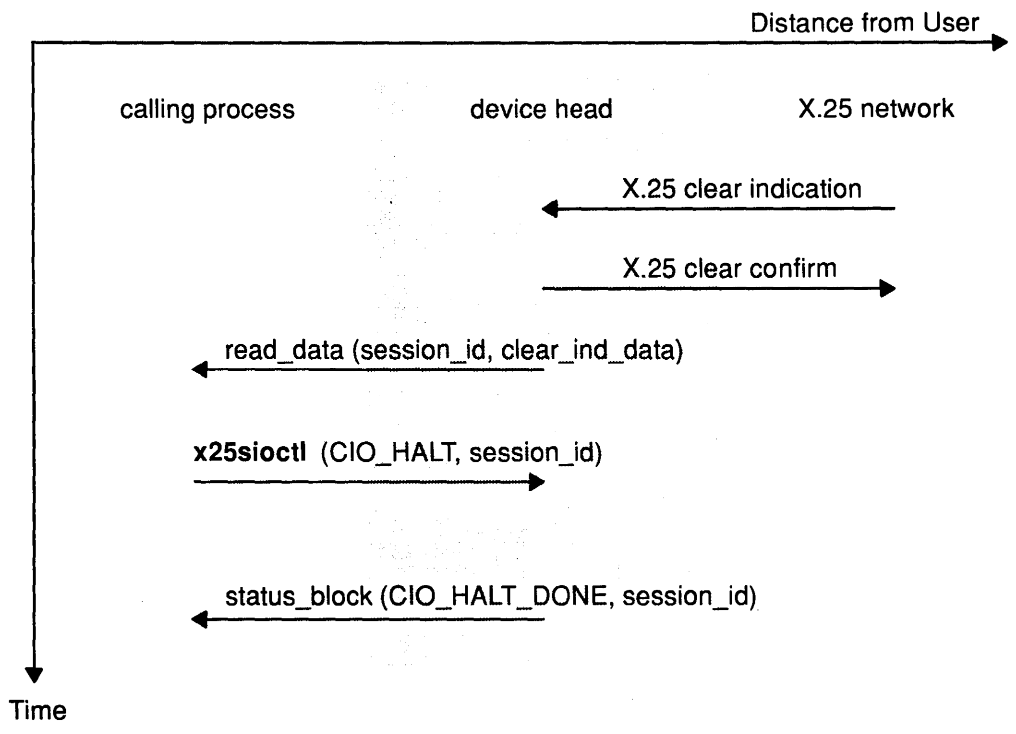
SESSION_SVC_LISTEN


Distance from User
calling process
device head
X.25 network
x25sioctl (CIO_HALT, session_id)
status_block (CIO_HALT_DONE, session_id)
Time

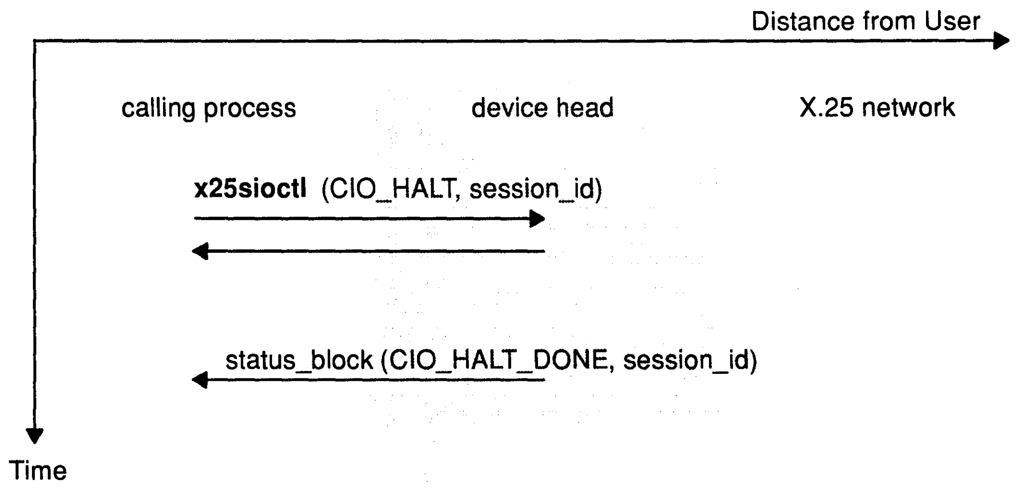
SESSION_SVC_IN, SESSION_SVC_OUT, locally-initiated clear


Distance from User
calling process
device head
X.25 network
x25sioctl (CIO_HALT, session_id, clear_request_data)
X.25 clear request
X.25 clear confirm
status_block (CIO_HALT_DONE, session_id, clear_confirm_data)
Time

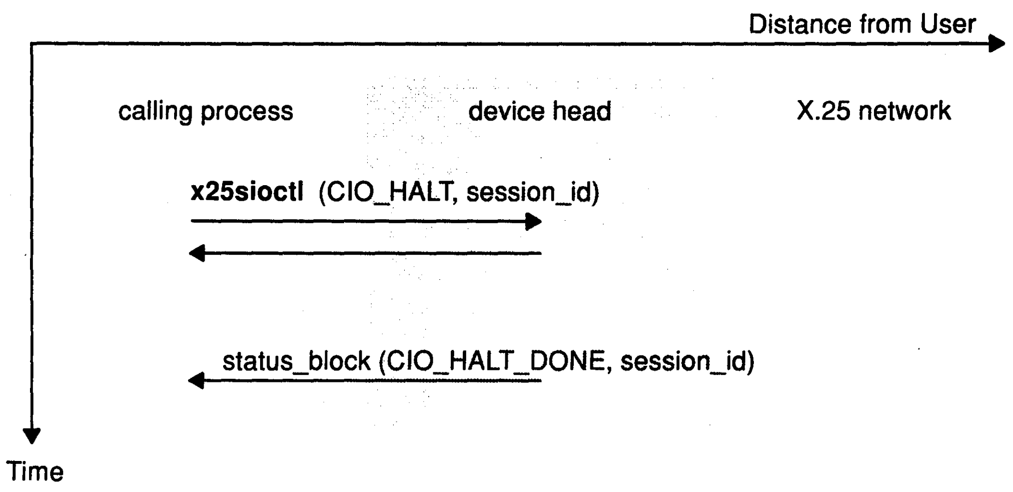
SESSION_SVC_IN, SESSION_SVC_OUT, remotely-initiated clear


Distance from User
calling process
device head
X.25 network
X.25 clear indication
X.25 clear confirm
read_data (session_id, clear_ind_data)
x25sioctl (CIO_HALT, session_id)
status_block (CIO_HALT_DONE, session_id)
Time

SESSION_PVC


Distance from User
calling process
device head
X.25 network
x25sioctl (CIO_HALT, session_id)
status_block (CIO_HALT_DONE, session_id)
Time

SESSION_MONITOR


Distance from User
calling process
device head
X.25 network
x25sioctl (CIO_HALT, session_id)
status_block (CIO_HALT_DONE, session_id)
Time

Data Transmission and Reception for the X.25 Device Handler

To transmit data the kernel-mode process issues an **fp_rwuio** kernel service. A pointer to an mbuf is passed as a parameter. Once the kernel-mode process has issued the **fp_rwuio** call, it must not access the mbuf(s) again. If the process has requested a transmit acknowledgment, the kernel-mode process can access the mbuf once the acknowledgment has occurred.

Depending on the options specified with the **fp_rwuio** kernel service, the device handler can call the **tx_fn** function specified by the **x25sopen** entry point to notify the kernel-mode process of transmit complete. The device handler's freeing of the mbuf depends on the options specified in the **fp_rwuio** call.

To transmit data for user-mode processes, an application uses a **writex** or **writevx** subroutine and specifies a buffer address. The device handler copies the buffer to a kernel space mbuf. Once the write returns to the application, the user can then access the buffer.

Data Reception for the X.25 Device Handler

For data reception, kernel-mode processes do not call an **fp_rwuio** subroutine. Instead, when data is received for a kernel-mode process's session, its **rx_fn** kernel procedure (specified at **x25sopen** time) is called, with each mbuf of data. It is the responsibility of the kernel-mode process to free the mbufs that contain the received data.

If a user mode application has an outstanding **x25sselect** entry point for Data Available, the device handler calls the **selnotify** kernel service to notify the application that data is available.

Common X.25 Device Handler Structures

The following structures are common to several X.25 device handler functions:

mbuf Structure

The **mbuf** structure is used by kernel-mode processes to transmit and receive buffers. The **mbuf** structure is defined in the **<sys/mbuf.h>** file. This structure contains the following fields:

m_next	Points to the next **mbuf** structure in a chain.
m_off	Identifies the offset data.

m_len	Identifies the length of the data.
m_type	Identifies the type of **mbuf** structure.
m_dat[MLEN]	Describes data storage on the buffer.
m_act	Links in a higher level list of **mbuf** structures.

Depending on the value of MLEN (the length of the mbuf data), the **mbuf** structure is used in one of the following ways:

- The amount of data required is less than MLEN. In this case, the **m_off** field contains the address of the **m_dat** field, and the **m_len** field contains the length of data in the **m_dat** field.

  Note: The **m_len** field must be in the range 0 to MLEN.

- The amount of data is larger than MLEN. In this case, the **m_off** field contains the address of a separate mbuf page cluster, the **m_len** field contains the length of that data, and the **m_dat** field is reserved.

If more buffers are required to hold the data, a series of **mbuf** structures can be chained together using the **m_next** pointer.

x25_buffer Structure

When an **mbuf** structure contains X.25 data, the **m_dat** field or the associated page cluster is defined by an **x25_buffer** structure. This structure is found in the **<sys/x25user.h>** file and contains the following fields:

pd Contains an **x25_packet_data** structure. This is the only field in the mbuf that contains data if the packet type is any of the following:

- PKT_RESET_REQ
- PKT_RESET_IND
- PKT_RESET_CONFIRM
- PKT_D_BIT_ACK
- PKT_INT_CONFIRM.

cd Contains an **x25_call_data** structure. This is valid only for clear-and-reset packets.

user_data Identifies unformatted user data. This is valid for PKT_DATA, PKT_INT, or PKT_MONITOR packet types.

x25_packet_data Structure

The **x25_packet_data** structure contains information about the X.25 packets being sent or received. This structure is found in the **<sys/x25user.h>** file and contains the following fields:

packet_type Defines the type of X.25 packet. The following are the available packet types:

- PKT_CALL_REQ
- PKT_INCOMING_CALL
- PKT_CALL_ACCEPT
- PKT_CALL_CONNECTED
- PKT_CLEAR_REQ
- PKT_CLEAR_IND
- PKT_CLEAR_CONFIRM

- PKT_RESET_REQ
- PKT_RESET_IND
- PKT_RESET_CONFIRM
- PKT_DATA
- PKT_D_BIT_ACK
- PKT_INT
- PKT_INT_CONFIRM
- PKT_MONITOR.

cause	Specifies the X.25 cause code. This is meaningful only for clear and reset packets.
diagnostic	Specifies the X.25 diagnostic code. This is meaningful only for clear and reset packets.
flags	Contains a bitwise OR of one of following values:

X25_Q_BIT	Qualifier bit. This bit is meaningful only for PKT_DATA packets. The qualifier bit is set to indicate the use of a higher level protocol within the data packet.
X25_D_BIT	The X.25 delivery-confirmation bit. This bit is set in a data packet to indicate that an end-to-end acknowledgment to the packet sequence is required. The delivery conformation bit should be set in the call-establishment packets to indicate that the D-bit may be used during this session.
X25_M_BIT	The X.25 more-data bit. This bit is meaningful only for data packets. For write operations, this bit is set by the caller to indicate that the block is a non-final element in a series of data blocks. In this case, the number of data bytes to transmit must be a multiple of the packet size.

For the **x25sread** entry point, the more Data bit is set if the buffer area supplied by the caller is too small for the complete packet sequence. Further data is returned in subsequent reads, until the final packet of the sequence is read. When the file packet is read, the more Data bit is cleared.

Note: The size of the buffers supplied by the caller need have no relationship to the X.25 packet size.

When the session protocol is PROTOCOL_YBTS, packetizing and packet assembly are done by the calling program, not by the X.25 device handler.

x25_call_data Structure

The **x25_call_data** structure is used to pass information about X.25 call setup and termination packets. This structure contains the following fields:

calling_address	Specifies a string of ASCIIZ decimal digits representing the calling X.25 address.
called_address	Specifies a string of ASCIIZ decimal digits representing the called X.25 address.
facilities_length	Specifies the number of bytes of optional data used for X.25 facilities. This cannot exceed 109.
user_data_length	Specifies the number of bytes of optional data used for user data. This cannot exceed 16.
optional_data	Contains the X.25 facilities and user data.

x25_diag_mem Structure

The **x25_diag_mem** structure is used for diagnostic reads and writes from the IBM X.25 Interface Co-Processor/2. This structure is found in the **<sys/x25user.h>** file and contains the following fields:

buffer	Points to the user buffer that is either going to be read from or written to.
card_page	Identifies the initial page of card memory that the read or write takes place from.
card_offset	Identifies the initial offset within the page.
length	Specifies the length of the user buffer.

x25_diag_io Structure

The **x25_diag_io** structure is used for diagnostic reads and writes from and to the I/O registers of the IBM X.25 Interface Co-Processor/2. This structure has the following fields:

register	Identifies the register that is to be written to or read from.
value	Specifies, in the case of a write, the value to be written. On a read, the value read is stored.

x25_diag_addr Structure

The **x25_diag_addr** structure is used to return the load page and offset that the diagnostic task is loaded at. This structure is found in the **<sys/x25user.h>** file and contains the following fields:

page	Identifies the page in the card's memory that the task was loaded into.
offset	Specifies the offset of the page the task was loaded into.

Related Information

The **fp_close** kernel service, **fp_open** kernel service, **fp_ioctl** kernel service, **fp_rwuio** kernel service, and **selnotify** kernel service.

The **write**, **writev**, **writex**, or **writevx** subroutine, **read**, **readv**, **readx** or **readvx** subroutine, **open** subroutine, **ioctl** subroutine.

Kernel Environment Programming on page 1–1.

POWERstation and POWERserver Hardware Technical Reference — Options and Devices.

SNA Services and the Receive Data Transfer Offset field in *Communications Programming Concepts.*

The 4–Port Multiprotocol Interface Adapter Technical Reference S33F–5337

The High Function Terminal (HFT) Subsystem

The High Function Terminal (HFT) subsystem is a collection of terminal-related device drivers unified around the concept of the virtual terminal. There are three device drivers in the HFT subsystem: the KTSM device driver, which controls the keyboard, tablet, sound device and mouse devices; the graphics input/output (GIO) device driver; and the virtual display device (VDD) driver. This chapter discusses the following topics:

- Screen Manager Ring
- Echo Maps
- Graphics Input/Output Devices
- Understanding the Virtual Display Device Driver
- Understanding Virtual Terminals
- Reading Input Data from a Ring Buffer
- Keyboard Send-Receive (KSR) Mode
- Data Stream Modes
- HFT Device Driver (HFTDD) User Interface
- How to Enter Monitor Mode
- How to Exit Monitor Mode
- Data Stream for HFT Virtual Terminals
- Keyboards
- Nonspacing Characters.

The HFT special file describes the read, write, and ioctl structures required to program the HFT subsystem.

HFT Subsystem Component Structure

A High Function Terminal (HFT) subsystem consists of several components:

- A screen manager, which keeps a list of open virtual terminals linked together in a group called the Screen Manager Ring.
- Virtual display drivers (VDDs). A VDD supports one particular type of display. An HFT subsystem can include up to four physical displays of various types.
- Virtual terminals (VTs). A virtual terminal supports the illusion that more devices exist than are physically present. Virtual terminals are logically independent of each other, but share physical resources over time. In other words, the HFT device driver is a multiplexed device. Currently, an HFT subsystem can support a maximum of 32 virtual terminals.
- The keyboard, tablet, sound, and mouse (KTSM) devices. A maximum of one of each device type can be attached with the keyboard/tablet/sound/mouse (KTSM) device driver.
- Software keyboards for keyboard-to-display symbol mapping.
- The graphics input/output (GIO) device driver, which supports lighted programmable function keys (LPFKs) and valuator, or dial, devices. A maximum of one of each device type can be attached with this device driver.

The Screen Manager Ring

Virtual terminals are linked together in a group called the *screen manager ring*. The screen manager places an entry in the ring for each virtual terminal opened. The terminal that is currently active is called the head of the ring. The last terminal on the ring is called the tail. When a new terminal is opened, that terminal becomes the head of the ring.

Three key sequences are used to switch between virtual terminals and to control which terminal is currently active. The active terminal is the terminal that accepts data from the input devices. Pressing the Alt + Action key sequence makes the next virtual terminal in the ring active. Pressing the Shift + Action key sequence on the active terminal makes the last virtual terminal active. When a command virtual terminal is set, pressing the command virtual terminal hot key activates the designated command virtual terminal. The command virtual terminal hot key is:

- The Control + Action key sequence on the keyboard
- Button four on the tablet
- The right *and* left buttons on the mouse.

Note that with three entries in the ring, any terminal can be accessed from any other with a single key sequence. With four or more entries, however, terminals may have to be skipped in order to activate a desired terminal. For example, in a ring with four terminal entries, the second terminal cannot be accessed directly from the fourth terminal (assuming that the fourth terminal is active). You must first skip to the first or third terminal.

Screen Manager Operations

These seven screen manager operations are available:

- Activating the virtual terminal.
- Hiding the virtual terminal.
- Setting the command virtual terminal.
- Restoring, or removing from hiding, the presence of a terminal in the ring.
- Enabling the command virtual terminal to be activated.
- Disabling the capability of the command virtual terminal.
- Disabling the capability of a virtual terminal to be activated.

Screen Manager Operations

There are seven screen manager operations.

Activating the Virtual Terminal

This operation places the virtual terminal specified by the **hf_vtid** field at the head of the screen manager ring. This position in the ring makes the virtual terminal the *active* terminal. The terminal's hidden flag is also cleared.

The screen manager cannot activate the specified virtual terminal if the currently active virtual terminal cannot be deactivated. The screen manager does not activate an untrusted terminal if the active terminal is trusted.

Hiding the Virtual Terminal

This operation logically removes terminals from the ring. Hiding a terminal causes it to be bypassed when its position in the ring would ordinarily make it the active terminal.

This operation marks the terminal identified by the **hf_vtid** field so that the screen manager does not activate it. This does not affect the terminal's position in the ring. When the hidden flag is set, the screen manager ignores the terminal's presence in the ring until an **SMUNHIDE** command is issued.

If the virtual terminal is active when the hide command is issued, the screen manager makes the terminal inactive (if possible). However, this does not prevent communication between the virtual terminal and the application running on it.

Hiding the active virtual terminal has the same effect that the last-window function does. That is, the previous terminal in the ring becomes active. If all virtual terminals are hidden, then the physical display continues to show the contents of the last virtual terminal that was hidden.

If the active terminal or the object terminal is trusted, the screen manager does not hide the object terminal.

Setting the Command Virtual Terminal

This operation designates a terminal as the command virtual terminal. The command virtual terminal is the terminal that is activated using the command window hot key: pressing the left and right buttons on the mouse simultaneously, pressing the number four button on the tablet puck, or pressing the Ctrl + Action key sequence.

Restoring (Unhiding) the Presence of a Terminal in the Ring

This operation restores the presence of the terminal in the screen manager ring. However, this action does not affect the ring position of the terminal or make it active. If the virtual terminal happens to be at the head of the ring when this command is issued, then it becomes visible and active. If the active terminal or the object terminal is trusted, the screen manager does not unhide the object terminal.

The **hf_vtid** field must contain the mpx of the virtual terminal where the command should be sent. The **hf_vtid** field is reserved.

Enabling the Command Virtual Terminal to Be Activated

This operation enables the command virtual terminal to be activated when the command window hot key is received. This is the default setting. Since all virtual terminals are affected, programs that change this setting should restore it as soon as the locator is no longer needed.

Disabling the Capability of the Command Virtual Terminal to Be Activated

This operation disables activation for a command virtual terminal when the command window hot key is received. The data reported is similar to that reported when a single button is pressed.

Disabling the Capability of a Virtual Terminal to Be Activated

This operation disables activation for a specific virtual terminal resulting from a hot key sequence. This virtual terminal is skipped in the ring as though it were hidden. However, it can be visible.

Echo Maps

An *echo map* is an array of bits that contains one bit for each possible keyed input. When keyboard input is entered, a keyboard translation process interprets the input as a particular display symbol or a particular control function.

The term *echo* can refer to either of two actions:

- Sending the character associated with a keystroke to the display screen.
- Performing the function associated with a control, and sending that information to the application.

The echo map is used to decide whether the result of this keyboard translation should be echoed (as defined above) or not. If the translation yields a display symbol, and the corresponding bit in the echo map is set to one, then the display symbol should be sent to the display screen. Alternatively, if the keyboard translation process yields a control function, (and the echo map bit is set), this function should be executed as if it were received in the data stream from the host.

For example, if you enter the character a, and its echo bit is 1, the character a is displayed at the current cursor position of the display screen. An echo bit set to 0 (zero) indicates the keyed display symbol or control is not echoed directly to the display until the information is received in an output data stream.

The structure of the echo map is described in Echo and Break Map Structure.

Graphics Input/Output Devices

The graphics input/output devices are lighted programmable function keys (LPFKs) and valuator, or dial, devices. Only one of each device type can be attached with the graphics input/output (GIODD) device driver.

Understanding the Virtual Display Device Driver

A Virtual Display Driver (VDD) supports one particular type of display. Up to four physical displays of specified types can be included in a High Function Terminal (HFT) subsystem.

The Virtual Display Driver is a software component containing a standard AIX operating system device driver plus routines that the HFT calls directly. These routines provide the HFT with a device-independent way of communicating with the display hardware.

You can perform the following display-related tasks with the **ioctl** and **write** operations:

ioctl Operation Options

- Adding a font: Use the Reconfigure (HFRCONF) **ioctl** Operation.
- Returning information about physical display IDs: Use the Query (HFQUERY) **ioctl** Operation.
- Returning information about a physical display: Use the Query (HFQUERY) **ioctl** Operation.
- Returning an ASCII data stream image of the current display screen: Use the Query (HFQUERY) **ioctl** Operation.

write Operation Options

- Redefining a virtual terminal's font palette: Use the Change Font Palette **write** Operation.
- Redefining a virtual terminal's cursor representation: Use the Redefine Cursor Representation **write** Operation.
- Redefining a virtual terminal's color palette: Use the Set KSR Color Palette **write** Operation.

Understanding Virtual Terminals

The *virtual terminal* concept supports the illusion that more devices exist than are physically present. Devices in a virtual terminal subsystem have characteristics and features not necessarily limited to those offered by the actual physical devices.

Virtual terminals are logically independent of each other but share physical resources over time. The virtual terminal that can accept physical input at a given time is called the *active* virtual terminal.

Virtual Terminal States

A virtual terminal can assume any of these six states:

Active	The virtual terminal that can accept input from the physical devices. When a virtual terminal becomes active, it becomes the head of the screen manager ring. If it is in KSR mode, it displays the cursor. You can use the Activate the Virtual Terminal **ioctl** operation to activate a virtual terminal.
Pseudoactive	A virtual terminal that is visible on a display screen but cannot take input from input devices. Output can be sent to this terminal.
Inactive	A virtual terminal that is not visible on any display screen and cannot take input from the input devices.
Command	This is the virtual terminal that is to be activated when a command window hot key is entered. You can use the Set the Command Virtual Terminal **ioctl** operation to designate a terminal as the command virtual terminal.
Trusted	The process that opened this virtual terminal is running a trusted shell, which restricts access to a virtual terminal by other processes.
Hidden	The virtual terminal is marked so that the screen manager does not activate it. This does not affect the terminal's position in the ring. However, the screen manager ignores the virtual terminal until it is changed from hidden.

Modes That Affect Virtual Terminals

There are three types of modes in the HFT subsystem:

Virtual Terminal modes

Virtual terminal modes define the type of terminal interface the application uses to access the device. There are two virtual terminal modes: KSR mode and MOM mode.

Data Stream modes

Data stream modes, relevant only in KSR mode, permit you to modify the display's presentation space. Protocol modes include display characteristics such as cursor placement after a line feed and echoing of keyboard input to the display.

Protocol modes

Protocol modes determine how the virtual terminal interprets, translates, and returns data. These modes also allow you to enable mouse, tablet, dial, or Lighted Programmable Function Key (LPFK) input, or to request untranslated keyboard input.

A virtual terminal in an HFT subsystem can be in either of two primary modes:

KSR Mode	Keyboard send-receive mode. In KSR mode, the virtual terminal emulates an ASCII terminal using ASCII data stream. In this mode, the virtual terminal has a presentation space (PS) of a fixed number of lines and columns. KSR mode is the default mode.
Monitor Mode	Monitor mode (MOM). In Monitor mode, applications have a direct output path to the display hardware and a shortened input path. Users can directly manipulate terminal display characteristics while in MOM mode.

Monitor Mode (MOM)

Monitor (MOM) mode of the virtual terminal is used to operate the display in alls-points-addressable mode. In MOM, the program sends output directly to the display adapter through a range of the memory-mapped I/O bus. Consequently, the program avoids **write** calls when updating the display screen.

A program can also read data from a circular buffer and avoid **read** operations. Some execution speed is gained by operating in MOM mode, but portability is sacrificed because the program depends on specific display adapters.

Notes: For Use of Monitor Mode

1. Do not leave the terminal open in Monitor mode.
2. Do not allow more than one process manipulating the request or grant to be open to a virtual terminal in Monitor mode.

You must use the specified **write** operation to switch from normal KSR mode to Monitor mode. The user program in Monitor mode participates in the next-window function by using subroutines to release the display temporarily. While the user program is active to the display, it performs output operations directly to the display hardware with memory-mapped I/O ports.

Valid ASCII Codes for MOM Mode

Only a subset of ASCII codes are valid for the **write** operation while in Monitor mode. All other codes are ignored. The valid operations are:

- Set Keyboard LEDs **write** Operation.
- Set LPFKs **write** Operation.
- Set Dial Granularities **write** Operation.
- Send Sound **write** Operation.
- Cancel Sound **write** Operation.
- Set Protocol Mode **write** Operation.
- Screen Request **write** Operation.
- Screen Release **write** Operation.

MOM Mode Tasks

The following procedures are available for working in MOM mode:

- How to Enter Monitor Mode.
- Requesting Screen Control.
- Reading Input Data from a Ring Buffer.
- Requesting Screen Release and Specifying an Input Ring Buffer.
- How to Exit Monitor Mode.

MOM Signals

These four signals are used in MOM mode:

SIGGRANT	Informs the user program that the display hardware can be directly accessed. This signal is sent following a Monitor mode Screen Request **write** call. It is also sent after the user has used the Next Window key to activate a Monitor mode terminal.
SIGRETRACT	Informs the user program that the display hardware must be released for use by another program. This signal is sent when a display screen is to be made inactive with the Next Window key. The proper response for the user program is to issue a Screen Release **write** call.
SIGKILL	Terminates all processes associated with a virtual terminal for which there has been no response to a **SIGRETRACT** signal within 30 seconds. The **SIGKILL** signal also closes the virtual terminal.
SIGMSG	Informs the user program that data has been placed into a previously empty input ring buffer.

Reading Input Data from a Ring Buffer

The input ring buffer is represented by the **hfmomring** structure, which is defined in the **<sys/hft.h>** file. The fields in the **hfmomring** structure are variables contained in the **hfmomring** structure. These fields are defined in **hft.h** File Structure for MOM **write** Operations.

Steps for Reading from the Input Ring Buffer

When a user program wants to read ring buffer input, it should first initialize the **hf_source** and **hf_sink** offset fields to make them equal. Initialization must occur before the screen request is issued. This indicates a buffer-empty condition. The program should then issue the **pause** subroutine and wait for input.

When the buffer goes from empty to not empty, the user program receives a **SIGMSG** signal. When this happens, characters can be extracted from the ring buffer, and the user should increment the **hf_sink** offset for each character extracted. It should also make sure to wrap around after reaching the end of the buffer.

Care should be taken to ensure that the buffer empty condition is properly detected. The program should test the equality of the offsets after each update of the **hf_sink** offset. Therefore, the order of operation is:

1. Extract a character.
2. Update the offset in its memory location.
3. Test the equality of offsets. If the offsets are equal, then set the **hf_intreq** field to 0xFF. (This resets the signal request.)

Detecting a Full Ring Buffer

If the value in the **hf_source** field equals the value (**hf_sink -1** (modulo ring size)), then the ring buffer is full. An overflow condition also exists if the value in the **hf_ovflow** field is 0xFF. The overflow condition indicates input data has been lost. The application can reset the overflow condition by clearing the **hf_ovflow** field.

Using the read Subroutine to Intercept Selected Keystrokes

Certain keys can be designated so they can be obtained using the **read** operation. The method for designating such keys is to set the break map bits for them to On. The Set Break Map (HFTSBREAK) **ioctl** Operation gives information on setting these break map bits.

Keyboard Send-Receive (KSR) Mode

A virtual terminal is in KSR mode by default. In this mode, it has a presentation space (PS) of a fixed number of columns and lines. A symbol can be placed at any column on any line in this presentation space. Graphics from the KSR data stream are placed in the PS relative to the cursor position. Keyboard input is also echoed relative to cursor position.

KSR Modes for Displaying Graphics

Two common modes for displaying graphics in KSR mode are Replace and Insert. An additional mode, AUTONL mode, determines cursor movement after the last column position of a line. The following are the three KSR modes:

Replace	In Replace mode, a graphic character sent to a KSR terminal is placed at the cursor, replacing the symbol already there.
Insert	In Insert mode, a graphic character sent to a KSR terminal is also placed at the cursor, but the symbol at the cursor and all symbols on the same line are shifted right one column position on the line. Characters shifted from the last column on the line are discarded.

AUTONL	Automatic new line mode (AUTONL) determines cursor movement after the last column position of a line. The AUTONL mode determines if the cursor wraps around to the first column position of the next line or stays at the last column on the current line.

If AUTONL mode is set, the cursor moves to the first column position of the following line. If the cursor is at the bottom line of the presentation space, the presentation space scrolls up one line. If AUTONL mode is reset, the cursor stays on the last column of the current line.

Blank lines in the presentation space and erased character positions are displayed in the active background color with normal attributes.

Tasks in KSR Mode

The following operations are available in KSR mode only:

- Specifying the KSR color palette to associate with specified display adapters: Set KSR Color Palette **write** operation.
- Redefining a virtual terminal's cursor representation: Redefine Cursor Representation **write** operation.
- Redefining a virtual terminal's font palette: Change Font Palette **write** operation.

Data Stream Modes

When a virtual terminal is in KSR mode, you can set the presentation space, or data stream mode, in either of two ways by:

- Issuing the SM (set mode) and RM (reset mode) multibyte controls.
- Setting the **ds_mode** field (data stream mode) of a **vtmdef** structure and issuing a Reconfigure (Default) **ioctl** Operation.

There are six types of presentation modes:

LNM	Linefeed/newline mode. If LNM mode is set, the line feed moves the cursor position to the first position of the next line. If LNM mode is reset, the line feed moves the cursor position down one line.
IRM	Insert/replace mode. If IRM mode is set, a graphic character sent to the display is placed at the cursor, replacing the symbol already there. If IRM mode is reset, a graphic character sent to the display is also placed at the cursor. However, all symbols at and to the right of the cursor on the same line are shifted right one column position. Characters shifted from the last column on the line are discarded.
SRM	Send/receive mode. If SRM mode is set, the virtual terminal does not echo the translated keyboard input. If SRM mode is reset and the echo map has been set correctly, then the virtual terminal echoes the translated keyboard input.

In most instances, echoing is done by the line discipline that the user selects. However, a provided echo map permits the HFT subsystem to echo specific characters. The echo map is a bit map with 512 bits corresponding to code points. If a bit is on, the code point is echoed.

The application should turn off the bits for the code points it does not want the HFT subsystem to echo. These characters usually include the escape character and the characters defined for Intr, Quit, and Erase. It can also include the characters defined for kill, EOF (end of file), EOL (end of line), pacing, and others.

TSM	Tabulation stop mode. If TSM mode is reset, then horizontal tabulation changes affect all lines. If TSM mode is set, horizontal tabulation changes affect only the line indicated by the cursor.
CNM	Carriage return/newline mode. If CNM mode is set, the carriage return causes the cursor to move to the first position of the next line. If CNM mode is reset, the carriage return moves the cursor position to the first character of the line indicated by the cursor.
AUTONL	Wrap to the next line when the end of line reached. If AUTONL mode is set, the cursor moves to the first column position of the following line. If AUTONL mode is reset, the cursor remains on the last column of the current line.

HFT Device Driver (HFTDD) User Interface

The High Function Terminal device driver (HFTDD) is the application/user interface to the High Function Terminal Subsystem (HFTSS). The following topics address HFT device driver user interface facilities:

- Understanding HFT Initial State
- Understanding select Support in the HFT
- HFT ioctl Operations
- Understanding HFT Output write Operations
- Reading Input with the read Operation
- Understanding Keyboard-Send (KSR) Mode
- Understanding Monitor (MOM) Mode.

Understanding HFT Initial State

The High Function Terminal (HFT) supplies default values for the following virtual terminal facilities:

- Keyboard-to-character mapping supplies a mechanism to take input from the keyboard device driver. The mapping then translates it to a character or ASCII control.
- Character-to-display symbol mapping supplies a default font used by the display to echo characters on the screen.
- Echo/break specification supplies a default map used by the HFT to echo characters on the screen and to break on none.
- Tab rack supplies default tabs at the first and last positions in the presentation space and every eighth position.
- Protocol mode flags supply a default of translating all keyboard input from the keyboard position to a character or ASCII control instead of sending the keyboard position to the operating system.

Default Values in the HFT

When a new terminal in the HFT subsystem is opened, it is initialized to a known state. These default values hold unless a redefinition is received from the application. The initial terminal state is the following:

Mode	Keyboard Send-Receive (KSR)
Echo Map	Echo all characters.

Although the echo map is set to echo all characters, the ASCII control SRM determines whether or not the HFT echoes characters. The default setting is to leave character echoing to the line discipline.

Break Map	Break for no characters.
Tab Rack	The first, every eighth, and the last position of every line.

The following are the available ASCII Controls:

– LNM	Not set
– IRM	Not set
– SRM	Set
– TSM	Not set
– CLM	Not set
– AUTONL	Set.

The following are ASCII controls available in Protocol mode:

– HFWRAP	Set
– HFMOUSE	Not set
– HFTABLET	Not set
– HFXLATKBD	Set
– HFHOSTS	Not set
– HFLPFKS	Not set
– HFDIALS	Not set
– HFJKANA	Not set.

Mouse Thresholds	2.75 millimeters horizontal, 5.5 millimeters vertical
Mouse Resolution	4 counts per millimeter
Mouse Sample Rate	60 samples per second
Mouse Scaling Factor	1:1
Tablet Dead Zones	0 millimeters horizontal, 0 millimeters vertical
Tablet Resolution	500 lines per inch
Tablet Sample Rate	1 sample per second
Tablet Origin	Lower left of the tablet
Tablet Conversion	English (versus Metric)
Font	Initially, and whenever the physical display device is changed, this is set to the default font for that display. All alternate fonts are initialized to the same default font.

The following is the default KSR Mode Color Palette:

Entry	**Color**
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	Gray
9	Light red
10	Light green
11	Light yellow
12	Light blue

Entry	Color
13	Light magenta
14	Light cyan
15	High intensity white.

select Operation Support in HFT

The High Function Terminal (HFT) device driver supports **select** operations in the following ways:

- Read **select** operations are satisfied when input data is available.
- Write **select** operations are always satisfied immediately.
- Exception **select** operations are never satisfied, or hang indefinitely if no time-out value is specified.

HFT Output write Operations

In the HFT Subsystem, the **write** operation can be used for both normal output and for control operations directed at the device. Using the **write** operation, ASCII data of any length can be sent to the virtual terminal. Virtual terminal control structures can also be sent to the virtual terminal using the **write** operation.

Each control structure is introduced by a virtual terminal data (VTD) character sequence. The VTD prefix consists of the ASCII codes ESC, [(left square bracket), and the `x` character (0x1B5B78). This prefix is followed by a 4-byte length (the **hf_len** field) and a 2-byte operation type code (the **hf_typehi** and **hf_typelo** fields). The data that follows this structure depends on the type of control.

The **hfintro** structure has the following format:

```
{
    char hf_esc;
    char hf_lbr;
    char hf_ex;
    char hf_len[4];
    char hf_typehi;
    char hf_typelo;
};
```

The **hf_len [4]** field is the total number of bytes in the header and associated data, excluding the 3-character VTD control sequence. The values of the **hf_typehi** and **hf_typelo** fields are described in:

- **hft.h** File Structures for General Output **write** Operations
- **hft.h** File Structures for KSR **write** Operations
- **hft.h** File Structures for MOM **write** Operations.

These three articles are discussed in the *Files Reference*.

Because the **hfintro** structure is an odd number of bytes in length, it is designated as the character array **hf_intro[HFINTROSZ]** in the structures that define the various operation requests. This prevents the C compiler from inserting bytes into the **hfintro** structure to align the subsequent fields on word boundaries. The **hf_typehi** and **hf_typelo** fields are sometimes referred to as the **hf_intro.hf_typehi** and **hf_intro.hf_typelo** fields.

All reserved and unused fields must be set to 0 (zero).

The three types of HFT write operations are as follows:

- **General Output write Operations**
  - Set Protocol Mode **write** Operation
  - Set Keyboard LEDs **write** Operation
  - Set LPFKs **write** Operation
  - Set Dial Granularities **write** Operation
  - Send Sound **write** Operation
  - Cancel Sound **write** Operation
  - Change Physical Display **write** Operation.
- **KSR write Operations**
  - Change Font Palette **write** Operation
  - Set KSR Color Palette **write** Operation
  - Redefine Cursor Representation **write** Operation.
- **MOM write Operations**
  - Screen Request **write** Operation
  - Screen Release **write** Operation.

Reading Input with the read Operation

Data read from an HFT device with the **read** operation can contain both character and noncharacter input.

Character Data

Character data is entered from the keyboard. The keyboard device driver processes keys pressed on the keyboard. If an application has requested untranslated keystrokes, the HFT subsystem reports them in an untranslated keystroke structure.

An untranslated keystroke structure contains all three of the following fields for each keystroke:

- The key position on the keyboard
- A scan code
- A status code containing the state of special keys (including Shift, Control, and Alternate Shift).

The untranslated keystroke structure (also called the **hfunxlate** structure) is described in **hft.h** File Structures for General **write** Operations.

Noncharacter Input

Noncharacter input can be read from several other sources in the HFT device. These sources are:

- Mouse
- Valuator dials
- Tablet
- Lighted Programmable Function Keys (LPFKs).

Data from non-keyboard devices is passed back from the **read** operation in the form of special control sequences. The input device control sequence reports input data from these devices.

Note: These control sequences contain binary data. To prevent the binary data from being misinterpreted as ASCII control codes, set the terminal's canonical processing to Off in the line discipline. You must also disable all signals.

To set the terminal's canonical processing to Off, you must set the **c_lflag** field to ISIG. For information about setting the **c_lflag** field to ISIG, see ISIG in **termio.h** File for AIX Version 2 compatibility and ISIG in **termios.h** File for POSIX compatibility.

The following input operations with the **read** operation are available:

- Untranslated Key Control **read** Operation.
- Input Device Report **read** Operation.

Protocol Modes

HFT protocol modes determine how the virtual terminal interprets, translates, and returns data. They allow the reporting of keyboard events such as shift key depression and input translation. In addition, they can enable non-keyboard devices, such as the mouse or tablet, to send data.

Setting Protocol Mode

You can set protocol modes in either of two ways by:

- Issuing a Set Protocol Mode **write** operation.
- Setting the **hf_select** field of a **vtmdef** structure and issuing a Reconfigure (Default) **ioctl** Operation.

Protocol modes are valid for both KSR and MOM mode unless otherwise indicated. If the mode is set to On (1 bit), the function is enabled. If the mode is set to Off (0 bit), the function is disabled.

Types of Protocol Modes

The following table lists the eight protocol modes in HFT. The mode-name literals are defined in the **<sys/hft.h>** file. The use of these literals with the **hfprotocol** structure is described in **hft.h** File Structures for General **write** Operations.

Mode	Valid	Meaning
HFHOSTS	KSR	A 0 (zero) bit (default) means not to report Shift-key depressions. A 1 bit means report Shift-key depressions. **HFHOSTS** mode specifies whether to report keyboard status changes. If **HFHOSTS** mode is set, the keyboard status information is returned in the KSI private ANSI control.
HFXLATKBD	KSR, MOM	A 1 bit (default) specifies that the keyboard input is translated. A 0 bit indicates send key data as untranslated key controls.
HFWRAP	KSR	A 1 bit (default) causes the cursor to wrap when the presentation space boundary is exceeded. A 0 bit specifies that the cursor should not wrap.
HFMOUSE	KSR, MOM	A 0 bit (default) disables the mouse from sending data. A 1 bit enables the mouse to send data.

Mode	Valid	Meaning
HFTABLET	KSR, MOM	A 0 bit (default) disables the tablet from sending data. A 1 bit enables the tablet to send data.
HFLPFKS	KSR, MOM	A 0 bit (default) disables LPFK input. A 1 bit enables LPFK input.
HFDIALS	KSR, MOM	A 0 bit (default) disables dial (valuator) input. A 1 bit enables dial input.
HFJKANA	KSR, MOM	A 0 bit (default) disables Kana shift state. A 1 bit enables Kana input (for use with Japanese license program only).

How to Enter Monitor Mode

Prerequisite Tasks or Conditions

1. The user must have a valid open file descriptor for a virtual terminal in Monitor mode (MOM).

Procedure

Entering MOM mode is a two-step process:

1. Issue the Enter Monitor Mode (HFSMON) **ioctl** operation to enter MOM mode. This step enables the **SIGGRANT** and **SIGRETRACT** Monitor mode signals. Issuing this operation also specifies the method by which processes are to receive the signals.
2. The program should set the preferred Monitor protocol modes. Only the following protocol modes are valid in MOM mode. Other modes are ignored.

- **HFXLATKBD**
- **HFMOUSE**
- **HFTABLET**
- **HFLPFKS**
- **HFDIALS**.

How to Exit Monitor Mode

Prerequisite Tasks or Conditions

1. The user must have a valid file descriptor of a virtual terminal in MOM mode.

Procedure

1. Write a screen release control (using the Screen Release **write** operation) and follow it with a KSR protocol control (the Set Protocol Modes **write** operation). This step is particularly important if the virtual terminal has been opened by another process. If the program is certain that no other processes have the terminal open, it can issue a close call to remove that virtual terminal.
2. Issue an Exit Monitor Mode **ioctl** operation to ensure that no Monitor mode signals have been sent to this process or other process in the terminal group in error.

Data Stream for HFT Virtual Terminals

Data stream includes all information that is sent to the HFT device driver with a **write** operation and is used in KSR mode only.

The data stream is composed of these three components:

- Code points (224 displayable)

- Single-byte control codes
- Multibyte control codes.

The AIX operating system native displays can address 256 distinct, displayable characters. The first 32 code points are reserved for control codes that do not have graphic representations. The remaining 224 code points denote distinct graphic characters. See Code Page for a schematic description of code points in the HFT data stream.

Nonspacing Characters in the KSR Data Stream

A *nonspacing character sequence* in the KSR data stream is a two-key sequence consisting of one of the 7 available diacritics followed by an alphabetic character or a space. This nonspacing or *dead* character facility is provided for convenience when typing diacritic (accented) characters. The HFT converts this two-key sequence into a single code point. The resulting character is the alphabetic character with the specified diacritic mark. A diacritic mark followed by a space translates to the diacritic character itself.

Valid Diacritic Characters

There are 7 valid diacritic characters:

Symbol	Function Code	Value
'	Acute Accent or Apostrophe	0xEF or 0x27
'	Grave Accent	0x60
^	Circumflex Accent	0x5E
¨	Umlaut Accent	0xF9
~	Tilde Accent	0x7E
°	Overcircle Accent	0xF8
Ç	Cedilla Accent	0xF7

Note: A cedilla accent is a curved accent that is suspended from a letter. In the preceding table, it is suspended from the letter C.

If a nonspacing character and the subsequent character do not combine to form a diacritic character in the set of predefined graphic symbols, then the diacritic mark is treated as a separate character code. For example, ~Q is treated as two characters: ~ and Q.

Multibyte Controls in Data Stream Data Overview

There are 52 basic multibyte sequences recognized by the virtual terminal in KSR mode. All of them begin with the ESC code (0x1B) although in control sequences, the ESC code is followed by a [(0x5B). These controls include all subsequent bytes up to and including the first code in the range 0x40 to 0x7F. In escape sequences, the ESC code is not followed by an [(0x5B). These controls include all subsequent bytes up to and including the first code in the range 0x30 to 0x7F.

Numeric parameters in control sequences contain no more than three digits. The numeric value of the parameter may be incorrect if more than three digits are used, and the numeric value never exceeds 255.

Controls affect a virtual terminal's presentation space (PS) and its related cursor (pointer into the PS). The presentation space is a logical array of display symbols and is N columns by M lines.

All multibyte sequences flow from the operating system or application to the terminal, with the following exceptions:

- KSI (Keyboard Status Information), PFK (Program Function Key report), VTL (Virtual Terminal Locator report), VTR (Virtual Terminal Raw keyboard report), VTK (Virtual Terminal Kanji status report), and VTA (Virtual Terminal Adapter report) all flow from the virtual terminal to the operating system or application.
- DSR (Device Status Report) and CPR (Cursor Position Report) can flow in either direction. The meaning of these controls depends upon the direction.

Invalid Multibyte Control Code Sequences

Invalid multibyte control sequences return an error Device Status Report to the program. Multibyte control sequences of more than 16 codes are considered invalid upon receipt of the 17th code. The 18th code is not considered a part of that sequence.

Categories of Valid Multibyte Control Code Sequences

There are six categories of valid multibyte control code sequences:

- Erasing Areas, Displays, Lines, and Fields
- Inserting and Deleting Lines and Characters
- Controlling Cursor Movement
- Clearing and Setting Tab Controls
- Performing Miscellaneous Tasks
- Scrolling.

Any multibyte control sequences not defined in these categories is ignored.

- **Erasing Areas, Displays, Fields, or Lines**

Mnemonic	Code Value
EA	Erase to end of the area, or from start of area, or all of area.
ED	Erase to end of display, or from start of the display, or all of the display.
EF	Erase to end of the field, or from start of the field, or all of the field.
EL	Erase to end of the line, or from start of the line, or all of the line.

- **Inserting and Deleting Lines and Characters**

Mnemonic	Code Value
DCH	Delete Character.
DL	Delete Line.
ECH	Erase Character.
ICH	Insert Character.
IL	Insert Line.

- **Controlling Cursor Movement**

Mnemonic	Code Value
CBT	Cursor Back Tab.
CHA	Cursor Horizontal Absolute.
CHT	Cursor Horizontal Tab.
CNL	Cursor Next Line.
CPL	Cursor Preceding Line.

CPR	Cursor Position Report.
CUB	Cursor Backward.
CUD	Cursor Down.
CUF	Cursor Forward.
CUP	Cursor Position.
CUU	Cursor Up.
CVT	Cursor Vertical Tab.
HVP	Horizontal and Vertical Position.
IND	Index.
NEL	Next Line.
RCP	Restore Cursor Position.
RI	Reverse Index.
SCP	Save Cursor Position.

- **Clearing and Setting Tab Controls**

Mnemonic	**Code Value**
CTC	Cursor Tab Stop Control.
HTS	Set Tab.
TBC	Clear Tab.
VTS	Set Tab.

- **Performing Miscellaneous Tasks**

Mnemonic	**Code Value**
DSR	Device Status Report Request.
DMI	Disable Manual Input.
EMI	Enable Manual Input.
KSI	Keyboard Status Information.
PFK	PF Key Report.
RIS	Reset to Initial State.
RM	Reset Mode.
SGR	Set Graphic Rendition.
SG0A	Set G0 Character Set.
SG1A	Set G1 Character Set.
SM	Set Mode.
VTA	Virtual Terminal Addressability.
VTD	Virtual Terminal Data.
VTL	Virtual Terminal Device Input.
VTR	Virtual Terminal Raw Keyboard Input.

- **Scrolling**

Mnemonic	**Code Value**
SD	Scroll Down.
SL	Scroll Left.
SR	Scroll Right.
SU	Scroll Up.

Single-Byte Controls in Data Stream Data Overview

Single-byte controls, also called control characters and control codes, have character values from 0 to 31 (0x00 to 0x1F). There are 25 single-byte controls that have no terminal functions. Eight single-byte controls have terminal functions.

Single-Byte Controls with Terminal Functions

Mnemonic	Function
BS	Backspace.
CR	Carriage Return.
DC1	Resume Output.
DC3	Suspend Output.
ESC	Escape.
FF	Form Feed.
HT	Horizontal Tab.
LF	Line Feed.
VT	Vertical Tab.

This table lists the mnemonic, code value, and function of each single-byte control code.

Table

Mnemonic	Code Values	Function	Description
BEL	0x07	Bell	Causes an audible alarm to sound.
BS	0x08	Backspace	Moves the cursor position to the left one column unless the cursor is at the left boundary of the presentation space. If the cursor is at the left boundary, the cursor position does not change.
CR	0x0D	Carriage Return	If the CNM mode is reset (default), the carriage return moves the cursor position to the first character of the line indicated by the cursor. If the CNM mode is set, the carriage return is treated as a NEL control code. It causes the cursor position to move to the first position of the next line. In this case, if the cursor is already on the last line of the PS, the PS lines scroll up one line. The top line of the PS disappears and a blank line is inserted as the new bottom line.
DC1	0x11	**Resume Output**	Resumes output that was suspended by an ASCII DC3 control.
DC3	0x13	Suspend Output	Temporarily suspends output.
ESC	0x1B	Escape	Defines the beginning of a multibyte control sequence.
FF	0x0C	Form Feed	Treated as a line end.

Table cont.			
Mnemonic	**Code Values**	**Function**	**Description**
HT	0x09	Horizontal Tab	Moves the cursor position forward to the next tab stop. If the cursor is already in the last column of a line, then the cursor position does not change. The CHT (Cursor Horizontal Tab) multibyte control performs a similar operation, but also performs line wrapping.
LF	0x0A	Line Feed	If the LNM mode is reset, the line feed moves the cursor position down one line. If the LNM mode is set (default), the line feed is treated as a NEL control code and moves the cursor position to the first position of the next line. In either case, if the cursor is already on the last line of the PS, the PS lines scroll up one line. The top line of the PS disappears and a blank line is inserted as the new bottom line.
VT	0x0B	Vertical Tab	Moves the cursor position down to the next line that is defined as a vertical tab stop. Tabs stops are always set at the first and last lines of the PS. If the cursor is already on the last line of the PS and HFWRAP mode is not set, the cursor stays on the last line in the PS. If HFWRAP mode is set, the cursor moves to the top line in the PS. The column position does not change.

Single-Byte Controls with No Terminal Functions

The following 23 single-byte controls have no terminal functions:

Mnemonic	Code Value	Function
NUL	0x00	Null
SOH	0x01	Start of Header
STX	0x02	Start of Text
ETX	0x03	End of Text
EOT	0x04	End of Transmission
ENQ	0x05	Enquiry
ACK	0x06	Acknowledge
SO	0x0E	Shift Out
SI	0x0F	Shift In
DLE	0x10	Data Link Escape
DC2	0x12	Device Control 2

Mnemonic	Code Value	Function
DC4	0x14	Device Control 4
NAK	0x15	Negative Acknowledgement
SYN	0x16	Synchronous
ETB	0x17	End of Block
CAN	0x18	Cancel
EM	0x19	End of Medium
SUB	0x1A	Substitute
SS4	0x1C	Single Shift 4
SS3	0x1D	Single Shift 3
SS2	0x1E	Single Shift 2
SS1	0x1F	Single Shift 1
DEL	0x7F	Delete

Keyboards

The AIX operating system supports these three natively attached keyboards:

- The 101-key keyboard
- The 102-key keyboard
- The 106-key keyboard.

Each of these keyboards differs slightly in its layout and function.

Key States

A software keyboard mapping table is maintained for each virtual terminal. This table maps a key position to an ASCII character, function, or string of characters. Each key on the keyboard has a numeric position code that is combined with the keyboard state when the key position is reported.

Available key states are:

- Base
- Shift
- Control
- Alternate
- Alternate Graphics
- Kana Base
- Kana Shift.

Each of the hardware keyboards can produce some, but not all, of these states.

One default software keyboard is selected at installation. Each virtual terminal that is opened operates with the selected mapping. A customized keyboard can be used as the system default after keyboard reconfiguration. You can use the HFSKBD **ioctl** subroutine to give each virtual terminal a different mapping.

Keys that Cannot be Remapped

The following keys are not redefinable because their function is predefined at the device-driver level.

Key Position	Function	States that Cannot be Remapped
30	Caps Lock key	All states
44	Left Shift key	All states
57	Right Shift key	All states
58	Control key	All states
60	Left Alternate key	All states
62	Right Alternate key	All states
64 Graphics	Action key	Shift, Control, Alternate, and Alternate
90	Num Lock key	Base and Shift states
133	Hiragana	All states.

Available Software Keyboard

There are 15 available software keyboards:

- Belgian-French/Dutch
- Canadian-French
- Danish
- Finnish/Swedish
- French
- German
- Italian
- Japanese
- Norwegian
- Portuguese
- Spanish
- Swiss-French/German
- UK English
- US English.

All keyboards, except the US English keyboard and the Japanese keyboard, have 102 keys. The US English keyboard has 101 keys, and the Japanese keyboard has 106 keys. The differences in the keyboards occur because each country requires a specific set of graphic characters (display set) for the native language as well as a specific keyboard layout (graphic key arrangement) on the keyboard.

Keyboard tables are shipped with the Base Operating System (BOS), and a keyboard table is installed during BOS installation.

Key Sequences

Most keying is done with either one-key or two-key sequences. For example, the a character is most often produced by one key (the A key) and the A character by two keys (Shift + A keys). If more than one state key is depressed when a character is keyed, (for example, Ctrl + Shift + A) only one state key affects the translation of the character. With the Ctrl + Shift + A sequence, the control state takes precedence over the shift state.

Three-key sequences have special meanings in the AIX operating system. The following keystroke combinations initiate the indicated system function. The notation Pad*n*, where *n* is a digit, indicates the *n* key on the numeric keypad to the right of the main keyboard area.

Note: Unless otherwise noted, the functions initiated by a three-key Ctrl + Alt + *key* sequence require the Alt key on the *left* side of the keyboard. Functions initiated with the Alt + *key* (or Shift + *key*) can be selected with either the left or right Alt key (or Shift key).

List of Special Key Sequences

There are three types of key sequences that have special meaning for the AIX operating system:

- Kernel Debugger

Ctrl + Alt + Pad4	Invokes the kernel debugger.

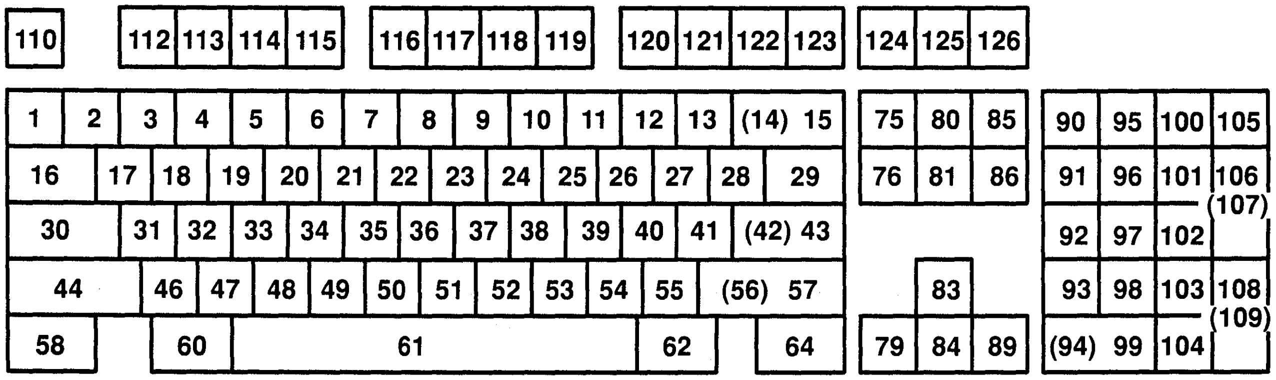
- System Dump Key Sequences

Note: Before attempting to use any of the following system dump key sequences, see Problem Determination Tips and Techniques.

Ctrl + Alt + Pad1	Performs a system dump to the primary device. ·
Ctrl + Alt + Pad2	Performs a system dump to the secondary device. Supports dumping to a logical volume or tape. Requires user intervention.

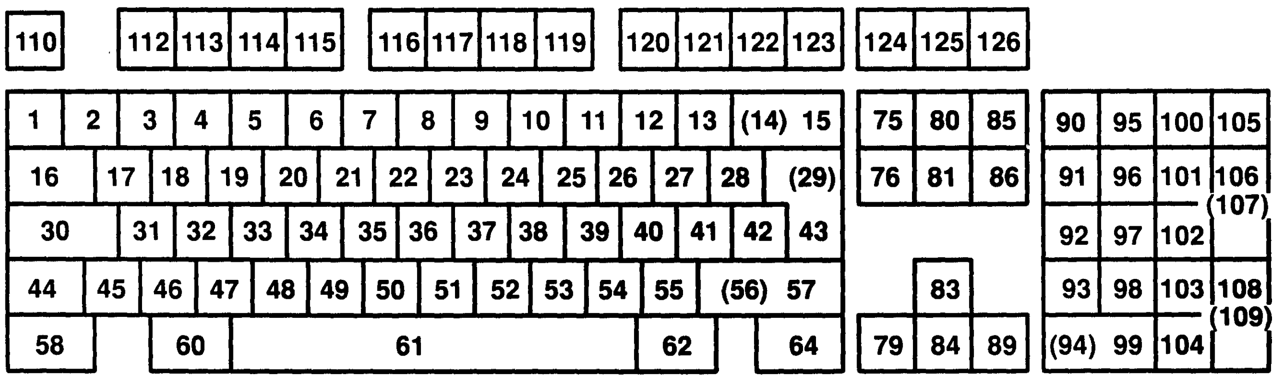
Keyboard Position Codes

These diagrams depict the key position codes for the 101-key US English keyboard and the 102-key keyboard.



US 101 Key Position Layout

Figure 48. 101-Key US English Keyboard Position Codes



WT 102 Key Position Layout

Figure 49. 102-Key Keyboard Position Codes

Keyboard States Overview

Each key on the keyboard has 7 potential states:

- Base
- Shift
- Ctrl (Control)
- Alt (Alternate)
- Alt Gr (Alternate Graphic)
- Kana Base
- Kana Shift.

U.S. Keyboard

For the US 101-key keyboard, the Alt Gr state is identical to the Alt state. The Kana base and Kana shift states are also identical to the base state. As a result, the US keyboard appears to have only four states:

- Base
- Shift
- Ctrl
- Alt.

The 102-key keyboard has these preceding states as well as Alt Gr.

Some of these keys are also governed by the Caps Lock key.

Japanese Keyboard

The Japanese keyboard does not have an Alt Gr key. Only the right Alt key is available. The Japanese keyboard has six states:

- Base
- Shift
- Control
- Alt
- Kana Base
- Kana Shift.

On keyboards that support Caps Lock, Caps Lock affects only keys whose Shift state yields the uppercase character (A, B, C) of the Base state lower-case character (a, b, c) of the key.

On keyboards that support Shift Lock, Shift Lock has the same effect as pressing a key while the Shift key is pressed.

Each key on the keyboard is assigned a unique 8-bit scan code that is sent when the key is pressed. The following table depicts key positions and their scan codes for the international character keyboards:

Key Positions and Their Scan Codes											
1	0x0e	23	0x3c	45	0x13	67	N/A	89	0x6a	111	N/A
2	0x16	24	0x43	46	0x1a	68	N/A	90	0x76	112	0x07
3	0x1e	25	0x44	47	0x22	69	N/A	91	0x6c	113	0x0f
4	0x26	26	0x4d	48	0x21	70	N/A	92	0x6b	114	0x17
5	0x25	27	0x54	49	0x2a	71	N/A	93	0x69	115	0x1f
6	0x2e	28	0x5b	50	0x32	72	N/A	94	0x68	116	0x27
7	0x36	29	0x5c	51	0x31	73	N/A	95	0x77	117	0x2f
8	0x3d	30	0x14	52	0x3a	74	N/A	96	0x75	118	0x37
9	0x3e	31	0x1c	53	0x41	75	0x67	97	0x73	119	0x3f
10	0x46	32	0x1b	54	0x49	76	0x64	98	0x72	120	0x47
11	0x45	33	0x23	55	0x4a	77	N/A	99	0x70	121	0x4f
12	0x4e	34	0x2b	56	0x51	78	N/A	100	0x7e	122	0x56
13	0x55	35	0x34	57	0x59	79	0x61	101	0x7d	123	0x5e
14	0x5d	36	0x33	58	0x11	80	0x6e	102	0x74	124	0x57
15	0x66	37	0x3b	59	N/A	81	0x65	103	0x7a	125	0x5f
16	0x0d	38	0x42	60	0x19	82	N/A	104	0x71	126	0x62
17	0x15	39	0x4b	61	0x29	83	0x63	105	0x84	127	N/A
18	0x1d	40	0x4c	62	0x39	84	0x60	106	0x7c	128	N/A
19	0x24	41	0x52	63	N/A	85	0x6f	107	0x7b	129	N/A
20	0x2d	42	0x53	64	0x58	86	0x6d	108	0x79	130	N/A
21	0x20	43	0x5a	65	N/A	87	N/A	109	0x78	131	0x20
22	0x35	44	0x12	66	N/A	88	N/A	110	0x08	132	0x28
										133	0x30

Nonspacing Characters Overview

A *nonspacing character sequence* is a two-key sequence consisting of one of the 7 available diacritical characters followed by an alphabetic character.

Among the available 224 graphic characters, 7 diacritics are used to construct diacritical, or accented, characters. The constructed diacritical characters yield a set of characters that exceed those engraved on any specific country-dependent keyboard.

Valid Nonspacing Character Sequences

Valid nonspacing character sequences are restricted to combinations of diacritical characters and alphabetic characters.

When the Virtual Terminal Mode Processor is running in translate mode, valid nonspacing character sequences are always folded into a single character before passing the keyboard input to the application.

A special case exists when the nonspacing character sequence consists of a diacritic followed by a space. In this case, the diacritic character itself is displayed and/or sent to the application.

A valid nonspacing character sequence causes a single accented character to be returned. A nonspacing character followed by a space character is a valid nonspacing character sequence. It returns the accent itself as a single character.

An example of a valid nonspacing character follows:

1st Key Pressed	2nd Key Pressed	Returned
Grave	z	e Grave (0x8a) – 1 character
Grave	Space	Grave Accent (0x60) – 1 character

Invalid Nonspacing Character Sequences

If the nonspacing character sequence is invalid, the HFT subsystem passes the nonspacing character to the application followed by the second character of the sequence. Invalid nonspacing character sequences include sequences that start with one of these three options:

- A nonspacing character followed by an alphabetic character (the resulting diacritical character does not exist in the system)
- A nonspacing character followed by a nonalphabetic character (numeric, control, function key)
- A nonspacing character followed by another nonspacing character.

An invalid nonspacing character sequence causes the accent character to be returned, followed by the code for the key pressed after the nonspacing key.

An example of an invalid nonspacing character follows:

1st Key Pressed	2nd Key Pressed	Returned
Grave	z	Grave Accent (0x60) – 2 characters z (0x7a)
Acute	PF1	Acute Accent (0xef) – 1 character PF1 (0x1b5b00000171)

An invalid nonspacing character sequence (*nonspacing character - nonspacing character*) causes the first nonspacing character of the sequence to be passed to the application. The next nonspacing character starts a new nonspacing character sequence.

File

/usr/include/sys/hft.h

Related Information

Query Screen Manager (HFTQSMGR) **ioctl** Operation, Request Screen Manager (HFTCSMGR) **ioctl** Operation, Set Echo Map (HFTSECHO) **ioctl** Operation, Reconfigure **ioctl** Operation, Enter Monitor Mode (HFSMON) **ioctl** Operation, Exit Monitor Mode **ioctl** Operation, Set Keyboard Map HFSKBD **ioctl** Operation.

The **tty** Special File and the **tty** Subsystem Overview in *Files Reference.*

The **hft.h** File Structures for Query **ioctl** Operations, **hft.h** File Structures for Special **ioctl** Operations, **hft.h** File Structures for **read** Operations, **hft.h** File Structures for General **write** Operations, **hft.h** File Structures for KSR **write** Operations, **hft.h** File Structures for MOM **write** Operations in *Files Reference.*

Screen Request **write** Operation, Screen Release **write** Operation, Set Protocol Modes **write** Operation.

The **tsh** Command.

The **select** subroutine.

termio.h File for compatibility with AIX Version 2 in *Files Reference.*

HFT Special File in *Files Reference.*

Logical Volume Subsystem

The following topics are available as guidance in understanding the logical volume subsystem:

- Physical volumes and the logical volume device driver
- The logical volume device driver
- Logical volumes and bad blocks.

Physical Volumes and the Logical Volume Device Driver

In a discussion of how the logical volume device driver (LVDD) interacts with physical volumes the following topics are relevant:

- Direct access storage devices (DASDs)
- Physical volumes
  - Implementation limitations
  - Reserved Sectors
- The logical volume device driver structure
- Interface to physical disk device drivers
- Logical volumes and bad blocks.

Direct Access Storage Devices (DASDs)

Direct access storage devices (DASDs) are *fixed* or *removable* storage devices. Typically, these devices are (hard) disks. A fixed-storage device is any storage device defined by the person who administers your system during system configuration to be an integral part of the system DASDs. The AIX Base 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 AIX operating system will *not* detect an error.

The following types of devices are *not* considered DASDs and are not supported by the logical volume manager (LVM):

- Diskettes
- CD-ROM (compact disk read-only memory)
- WORM (write once read mostly).

DASDs Device Block-Level Introduction

The DASD *device block* (or *sector*) level is the level at which a processing unit may request low-level operations on a device block address basis. Typical low-level operations for DASD are `read-sector`, `write-sector`, `read-track`, `write-track`, and `format-track`.

A DASD stores data in a way that allows for its rapid retrieval from random addresses as a stream of one or more blocks. Many DASDs perform best when the blocks to be retrieved are close (in physical address) to each other.

DASDs consist of a set of flat, circular, rotating platters. Each platter has one or two sides on which data is stored. Platters are read by a set of nonrotating, but positionable, read or read/write *heads* that move together as a unit. The following are terms used when discussing DASD device block operations:

sector	A contiguous, fixed-size block of data on a DASD. To maintain compatibility with the traditional UNIX(TM) model of DASD, every sector of every AIX DASD is defined to be exactly 512 bytes.
track	A track is a contiguous set of sectors on a single DASD. A track corresponds to the surface area of a single platter swept out by a single head while the head remains stationary. An AIX DASD contains at least 17 sectors per track. Otherwise, the number of sectors per track is not defined architecturally and is device-dependent. A typical AIX DASD track can contain 17, 35, or 75 sectors. An AIX DASD might contain 1024 tracks. The number of tracks per DASD is not defined architecturally and is device-dependent.
head	A head is a positionable entity that can read and write data from a given track located on one side of a platter. Usually a DASD has a small set of heads that move from track to track as a unit. There must be at least 4 heads on a DASD. Otherwise, the number is not defined architecturally and is device-dependent. A typical DASD might have 8 heads.
cylinder	The path swept out on the entire set of platters that can be read or written by the set of heads (when stationary). This path is called a cylinder. If a DASD has *n* number of vertically aligned heads, a cylinder is composed of *n* number of vertically aligned tracks.

Physical Volumes

A physical volume is a DASD structured for *physical level* requests. The physical level 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`.

The following are terms used when discussing DASD volumes:

block	A contiguous, 512-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 both 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 DASD devices (for example, SCSI, ESDI, or IPI) that may be placed in a given volume group.

Note: A given physical volume must be assigned to a volume group before that physical volume may be used by the AIX Base Operating System.

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 1016 physical partitions per physical volume
- The partition size is restricted to 2***n* bytes, for 20 <= *n* <= 28.
- The physical block size is restricted to 512 bytes.

Physical Volume Layout

A physical volume consists of a logically contiguous string of physical sectors. Sectors are numbered 0 through *LPSN*, where *LPSN* is the last physical sector number 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 is 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 **<sys/hd_psn.h>** file describes the information stored on the reserved sectors. In this file, the locations of the items in the reserved area are expressed as physical sector numbers 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*, and the *LVM 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 **<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 is assigned by the manufacturer of the physical volume. However, if a physical volume is part of a volume group the **pv_id** field may 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 **<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 **<lvmrec.h>** file.

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 contains the volume group descriptor area/volume group status area and follows the first 128 reserved sectors. The other is an area 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 M-byte physical volume with a partition size of 1 M-byte has 200 partition map entries.

- A name list

  This list contains the the special file names of each logical volume in the volume group.

- A volume group trailer

  This trailer contains an ending timestamp for the volume group descriptor area.

When a volume group is varied online, at least two readable copies of the volume group descriptor area are necessary in order to perform recovery operations. (The *vary-on* operation, performed by using the **varyonvg** command, makes a volume group available to the system.)

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 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 all volumes containing a volume group descriptor area must be able to come online before the vary-on operation is considered successful. A majority ensures that at least one copy of the volume group descriptor areas used to perform recovery was also one of the volume group descriptor areas used during the previous *vary-off* operation. If this is not the case, the consistency of the volume group descriptor area cannot be insured.

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.

Character I/O requests are performed by issuing a read or write 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 mirroring and bad-block relocation) 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 **<sys/buf.h>** file and contains all needed information about an I/0 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 physical **buf** structure (**pbuf**) is defined in the **<sys/dasd.h>** file. It is a standard **buf** structure with some additional fields. These fields are used by the LVDD 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 Logical Volume Device Driver

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:

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 logical volume device driver.
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. When a character request spans partitions or logical tracks (32 - (4K pages)), the LVDD **ddread** routine breaks it into multiple requests. The

routine then builds a buffer for each request, and passes it to the LVDD **ddstrategy** entry point, which handles logical block I/O requests.

If the *ext* parameter is set (called by **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 IOCINFO and XLATE operations, which return LVM configuration information.

Bottom Half of Logical Volume Device Driver

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. The bottom half of the LVDD is divided into three layers as follows:

- Strategy
- Scheduler
- Physical.

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.

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 mirror write consistency cache. For each logical request the scheduler layer schedules one or more physical requests. This involves 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 mirror write consistency cache for the volume group. If a logical volume is using mirror write consistency (MWC), 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.

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 thus hidden from the other two layers.

Interface to Physical Disk Device Drivers

Physical disk device drivers should adhere to the following criteria if they are to be accessed by the logical volume device driver:

- 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 on (defined in the **<sys/buf.h>** file) in the **b_flags** field.
  - The **b_error** field to E_MEDIA (defined in the **<sys/errno.h>** file).
  - The **b_resid** field to contain 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 fails, 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 should set the following:
  - The **b_error** field should be set to ESOFT (defined in the **<sys/errno.h>** file).
  - The **b_flags** field should have the **B_ERROR** flag set on
  - The **b_resid** field should be 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 **<sys/lvdd.h>** file) set on in the **b_options** field. This indicates that the device driver is to do a hardware relocation on this request. If the device driver does not support hardware relocation the following should be set:
  - The **b_error** field should be set to EIO (defined in the **<sys/errno.h>** file).
  - The **b_flags** field should have the **B_ERROR** flag set on.
  - The **b_resid** field should be set.
- 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 **<sys/lvdd.h>** file.

Logical Volumes and Bad Blocks

The physical layer of the LVDD initiates all bad-block processing and isolates all of the decision making from the physical disk device driver. This is done so that the physical disk device driver does not need to know anything about *mirroring.* Mirroring is the duplication of a physical partition that contains data.

Relocating Bad Blocks

The physical layer of the logical volume device driver (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 three separate pieces. The first piece contains any blocks up to the bad block. The second 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 bad block to the end of the request. These separate pieces are processed sequentially.

Once the I/O for the first of the separated 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 remaining (third) piece. Once 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 the block, reading the mirror, and writing the contents of the good mirror to the relocated block. With mirroring, the user need not even know when bad blocks are found. However, the physical disk device driver does in fact 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 non-mirrored 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 it 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, an I/O error is returned to the original requestor.

If the operation was for a mirrored read, 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.

Related Information

The **buf** structure.

The **lvdd** special file.

The **write** subroutine, **readx** subroutine.

The **iodone** kernel service, the **bread** kernel service, **bwrite** kernel service.

Communication I/O Subsystem on page 8–1.

SCSI Subsystem on page 12–1.

Bad Block Relocation Policy in *General Programming Concepts*.

The Vary-On and Vary-Off Process in *General Programming Concepts*.

Understanding Volume Groups in *General Programming Concepts*.

Device Driver Classes on page 2–1,and Device Driver Roles on page 2–2, Device Driver Structure on page 2–3.

Logical Volume Storage Overview in *General Programming Concepts*.

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 AIX operating system and new printer types. Printers Provided with AIX Version 3 lists printers that are already supported. This chapter covers the following topics:

- Adding a New Printer Type to Your System
- Adding a Printer Definition
- Adding a Printer Formatter to the Printer Backend.

Printer Types Currently Supported by IBM

To configure a supported type of printer, you need only to run the **mkvirprt** command to create a Customized printer file for your printer. This Customized printer file, which is in the **/usr/lpd/pio/custom** directory, describes the specific parameters for your printer.

Printer Overview for System Management discusses how to add, delete, and change printers and print queues in the system.

Printer Types Currently Unsupported by IBM

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.

Refer to these two examples:

- Example of the Simple Addition of a New Printer describes how to add a printer that is similar to an already supported printer.
- Example of the Complex Addition of a New Printer describes how to add a printer that does not closely resemble an already defined printer type.

Adding a New Printer Type to an AIX System outlines generic instructions for adding either type of printer. How to Add an Undefined Printer lists specific steps for adding a new type of printer to your RISC System/6000.

Adding an Unsupported Device to the System 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.

Adding a New Printer Definition

There are two ways of adding a new Printer definition to the Predefined printer directory. If you are making only relatively simple modifications, you can edit an existing printer definition to create a Customized Printer definition for your new printer. See How to Add an Undefined Printer for information about making these minor changes.

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 do this.

Steps for adding a new printer-specific formatter to the printer backend are discussed in Adding a Printer Formatter to the Backend.

Note: These instructions apply to the addition of a new printer definition to the system, not to the addition of 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 AIX operating system, you must also provide a new device driver. Device Driver Introduction provides guidance on writing your own device driver.

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. Since 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/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 enumerate 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 by IBM all along. Specific information about this printer can then be extended, added, modified, or deleted, as necessary.

Printers Provided with AIX Version 3 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 AIX 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 back end. If a new backend is required, see Understanding the Printer Backend.

Subroutines for Print Formatters

The **pioformat** formatter driver provides these six subroutines for the print formatters that it loads, links, and drives:

piocmdout	Outputs an attribute string for a printer formatter.
pioexit	Exits from a printer formatter.
piogetstr	Retrieves an attribute string for a printer formatter.

piogetopt	Used by printer formatters to overlay default flag values from the database with override values from the command line.
piogetvals	Initializes a copy of the database variables for a printer formatter.
piomsgout	Sends a message from a printer formatter.

The **pioformat** formatter driver requires a print formatter to contain these five function routines:

initialize	Performs printer initialization.
lineout	Formats a print line.
passthru	Passes through the input data stream without modification or formats the input data stream without assistance from the formatter driver.
restore	Restores the printer to its default state.
setup	Performs setup processing for the print formatter.

Related Information

The **mkvirprt** command.

Printer Overview for System Management in *General Concepts and Procedures*.

Writing a Device Driver on page 2–1.

Printers Provided with AIX Version 3, Printer Overview for System Management, Understanding the Printer Backend, How to Add an Undefined Printer in *General Concepts and Procedures*.

The SCSI Subsystem

The following topics are available as guidance in understanding the SCSI subsystem:

- The sc_buf Structure
- Execution of I/O Requests
- SCSI Device Driver Internal Commands
- SCSI Error Recovery
- Required SCSI Adapter Device Driver ioctl Commands
- A Typical SCSI Driver Transaction Sequence
- Other SCSI Design Considerations.

The following information about SCSI device and adapter device drivers is available in the *Calls and Subroutines Reference*:

- The SCSI **cdrom** Device Driver
- The SCSI **scdisk** Device Driver
- The SCSI **rmt** Device Driver
- The SCSI Adapter Device Driver.

Introduction

This section is intended to describe the AIX interface between a SCSI device driver and a SCSI adapter device driver. The primary audience for this section consists of those wishing to design and write a SCSI device driver that interfaces successfully with an existing AIX SCSI adapter device driver. A second audience consists of those wishing to design and write a SCSI adapter device driver that interfaces successfully with existing SCSI device drivers.

Terminology

This section frequently refers to both a *SCSI device driver* and a *SCSI adapter device driver.* These are two distinct AIX device drivers working 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 purpose of the SCSI adapter device driver (the lower layer) is to provide the software interface to the system hardware. This hardware includes the SCSI bus hardware and any other system I/O hardware required to execute an I/O request. The SCSI adapter device driver should hide the details of the I/O hardware from the SCSI device driver, allowing the upper driver to be written with as little knowledge of the system hardware as possible. The software interface to this layer of code is designed with this in mind.

The SCSI adapter device driver handles everything related to managing the SCSI bus, but has only general knowledge of SCSI devices. For example, it knows what a SCSI command is, and how to send it, but knows nothing of the contents of the SCSI command being sent. Knowledge of device specifics are left to the upper layer. This interface allows the upper driver to talk with different SCSI bus adapters without requiring special code paths for each adapter. The SCSI adapter device driver also provides recovery and logging for errors related to the SCSI bus and system I/O hardware.

Responsibilities of the SCSI Device Driver

The purpose of the SCSI device driver (the upper layer) is to provide the rest of the AIX operating system with the software interface to a given SCSI device (or class of SCSI devices). It knows what 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. However, the SCSI device driver knows nothing about how to manage adapter resources or how to give the SCSI command to the adapter. Knowledge of adapter and system specifics 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 AIX operating system provides several kernel services that allow the SCSI device driver to make calls to SCSI adapter device driver entry points without knowing about the actual name or address of those entry points. The logical file system kernel services can provide more information.

General Information

The interface between the SCSI device driver and the SCSI adapter device driver 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 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.

Further information about the interface between SCSI device and adapter device drivers is available:

- The **sc_buf** Structure
- Execution of Spanned and Unspanned SCSI I/O Requests
- SCSI Error Recovery
- SCSI Device Driver Internal Commands
- Requirements for SCSI ioctl Operations
- A Typical SCSI Driver Transaction Sequence
- Other SCSI Design Considerations.

Information about the following SCSI device and adapter device drivers is available:

- The SCSI **cdrom** device driver
- The SCSI **scdisk** device driver
- The SCSI **rmt** device driver
- The SCSI adapter device driver.

The sc_buf Structure

The **sc_buf** structure is used for communication between the SCSI device driver and the SCSI adapter device driver for a particular 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 **<sys/scsi.h>** header file.

Fields in the **sc_buf** structure are to be used as follows:

- Reserved fields should be set to a value of 0, except where noted otherwise.
- The **bufstruct** field contains a copy of the AIX 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 the use of the SCSI adapter device driver, if needed. The current definition of the **buf** structure can be found in the **<sys/buf.h>** include file.
- 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 non-spanned transfer. A 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.
- The **scsi_command** field, defined as a **scsi** structure, contains the SCSI ID, 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 decimal 6,10, or 12.
  - The **scsi_id** field is the SCSI physical unit ID.
  - 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.

    For normal use, the SC_NODISC bit should not be set, as it allows a device, while executing commands, to monopolize the SCSI bus, possibly to the detriment of other devices and the system. The SC_NODISC bit may be set when it is desirable for a particular device to keep control of the bus once it has successfully arbitrated for it. This may be true when this is the only device on the SCSI bus (or the only device that will be in use for a period of time). For performance reasons, it may not be desirable to go through SCSI re-selections to save SCSI bus overhead on each command.

    For 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. Since other errors may also result in the **SC_SCSI_BUS_FAULT** flags being set, the SC_ASYNC bit should only be set on the last retry of the failed command.

  - 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 opcode and logical unit identified individually. The **sc_cmd** structure contains the following fields:
    - The **scsi_op_code** field is the standard SCSI opcode for this command.
    - The **lun** field is the standard SCSI logical unit (0-7) for this physical SCSI device controller. Typically, there will be one LUN per controller (LUN=0) for devices with imbedded controllers. Note that the actual LUN ID is only the upper three bits of this byte.
    - The **scsi_bytes** field is the remaining command-unique field of the SCSI command block. The actual number of bytes in this field depends on the vaule in the **scsi_op_code** field.
- The **timeout_value** field is the timeout limit (in seconds) to be used for completion of this command. A timeout value of 0 means no timeout should be applied to this I/O request.
- The **status_validity** field contains the following bit flags:

SC_SCSI_ERROR
The **scsi_status** field is valid.

SC_ADAPTER_ERROR
The **general_card_status** field is valid.

Note: The following order of precedence should be followed by SCSI device drivers when analyzing the returned status:

- If the **sc_buf.bufstruct.b_flags** field has the B_ERROR flag set, then some 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, two interpretations are possible. Either the command simply needs to be restarted, or it was canceled at the request of the SCSI device driver.

  If the **b_error** field is found to contain the EIO value, then either one or no flag will be set in the **sc_buf.status_validity** field. If some flag has been set, then an error in either the **scsi_status** or **general_card_status** field is being reported.

  If the **status_validity** field is zero, then the **sc_buf.bufstruct.b_resid** field should be examined to see if this command was, in fact, in error. There are many cases where the **b_resid** field is nonzero and no error has occurred. The SCSI device driver must evaluate the **b_resid** field with regard to the SCSI command being sent and the SCSI device being driven to decide whether an error has occurred.

- If the **sc_buf.bufstruct.b_flags** field does not have the B_ERROR flag set, then no error is being reported. However, even in this case, 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 may still not represent an error. In this case, the SCSI device driver must decide if an error has occurred.

  If a nonzero **b_resid** field is found to represent an error condition, the SCSI device driver must be aware that the device queue has not been halted by the SCSI adapter device driver. Note that here it is possible for one or more succeeding queued commands to be sent to the adapter (and possibly the device). Recovering this situation is the responsibility of the SCSI device driver.

- In any of the above cases, if **sc_buf.bufstruct.b_flags** 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 to continue operations) must have SC_RESUME set in the **sc_buf.flags** field.

• The **scsi_status** field (in the **sc_buf** structure) 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 any time the **scsi_status** 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.

• The **general_card_status** field is valid when its **status_validity** bit is nonzero. The **sc_buf.bufstruct.b_error** field should be set to EIO any time the **general_card_status** field is valid. This field contains generic SCSI adapter card status and is intentionally general in coverage so that it can report error status from any typical SCSI adapter. Some of these error conditions are indicative of a SCSI device failure. Others are SCSI bus- or adapter-related.

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 it should generally be processed or recovered, or both, by the SCSI adapter device driver.

If the error is recovered successfully by the SCSI adapter device driver, then 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, then 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 generally 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 while the SCSI device driver logs SCSI device-related errors. In the following description, a capital letter A after the name indicates that the SCSI adapter device driver handles error logging. A capital letter H indicates that the SCSI device driver handles error logging.

SC_HOST_IO_BUS_ERR (A)	The system I/O bus has generated or detected an error during a DMA transfer.
SC_SCSI_BUS_FAULT (H)	The SCSI bus protocol or hardware has failed.
SC_CMD_TIMEOUT (H)	The command involved timed out before completion.
SC_NO_DEVICE_RESPONSE (H)	The target device would not respond to selection phase.**S**
C_ADAPTER_HDW_FAILURE (A)	The adapter is indicating an onboard hardware failure.
SC_ADAPTER_SFW_FAILURE (A)	The adapter is indicating microcode failure.
SC_FUSE_OR_TERMINAL_PWR (A)	The adapter is indicating a blown terminator fuse or bad termination.
SC_SCSI_BUS_RESET (A)	The adapter is indicating that the SCSI Bus has been reset.

- The **flags** field contains bit flags sent from the SCSI device driver down 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 transaction, 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 specifed 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.

Execution of 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 down 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 will either recover any errors on the transaction or return through **iodone** to the calling level.

The SCSI device driver must 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.

When the transactions are passed from the SCSI device driver down to the SCSI adapter device driver, they are placed in the SCSI adapter device driver's own queues. As they are processed, the SCSI adapter device driver returns the transaction through the **iodone** kernel service to the SCSI device driver. The SCSI adapter device driver must always process requests in the order received.

Spanned (Consolidated) Commands

Some kernel operations may be composed of sequential operations to a device. One example would be writing consecutive blocks to disk. These may or may 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/gather operations required, **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 (through the **buf.av_forw** field), giving the SCSI adapter device driver enough information to perform the DMA scatter/gather required. This includes at least the buffer starting address, its length, and the buffer's cross memory descriptor.

The spanned requests should always be for requests in either the read or write direction, but, of course, not both, as the SCSI adapter device driver must be given a single SCSI command to handle the requests. The spanned request should always be made up of complete I/O requests (the additional **struct bufs**). 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 may 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 **<sys/scsi.h>** header 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 should only consolidate commands that are memory page aligned at both their starting and ending addresses. Specifically, this applies to the consolidation of *inner* memory buffers. That is, 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 need not be memory page aligned.

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 size must be broken up by the SCSI device driver. Note that for this case, known as a *fragmented command*, the **sc_buf.bp** should be NULL so that the SCSI adapter device driver knows it should use only the information in the **sc_buf** structure to prepare for the DMA operation.

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.

There are no special requirements for commands with short data phase transfers (less than or equal to 256 bytes), other than that the SCSI device driver must have pinned the memory being transferred into or out of. 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 consideration 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 of this consideration, a SCSI device driver that initiates an internal command with more than 256 bytes must have preallocated and pinned an area of some multiple of the system page size. It must not then 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's **iodone** routine is called for the transaction (and any other transactions to those pages).

SCSI Error Recovery

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 properly recover from this condition. After this recovery, it should reschedule the transaction that had the error. In many cases, the SCSI device driver need only retry the failed operation.

The SCSI adapter device driver should never retry a SCSI command on error after the command has successfully been given to the adapter. Certain devices's commands cannot be retried immediately after a failure (for example, tapes and other sequential access devices). If such an error occurs, the command should be failed with appropriate error status and returned through 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.

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 may be required in the SCSI adapter device driver to support, for example, system management facilities and diagnostics.

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 **<sys/devinfo.h>** header file. The SCSI device driver should request the IOCINFO **ioctl** operation (probably during its open routine) to get the adapter's maximum transfer size.

For each of the **ioctl** operations described here, the *arg* parameter 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 2 bytes are reserved and should be set to zero.) This provides the information required to allocate or deallocate resources and perform SCSI bus operations for the **ioctl** operation requested.

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** 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 abort an operation instead of waiting for completion or timeout. The SCIORESET command is provided for clearing device hard errors and competing initiators's reservations during open processing by the SCSI device driver.

Note: The SCSI adapter device driver **ioctl** operations can only be called from the process level. They cannot be executed from a call on any more favored priority level. Attempting to call them from a more favored priority level can result in a system crash.

The following information is provided on the various operations:

- **SCIOSTART**

  This operation allocates and initializes SCSI device-dependent information local to the SCSI adapter device driver. This should be run only on the first open of an ID/LUN device. Subsequent SCIOSTART commands to the same ID/LUN will fail unless an intervening SCIOSTOP command is issued.

  The following values for the **errno** global variable should be 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 are the same SCSI ID as the adapter's.

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 will fail.

  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.

- **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 causes the SCSI adapter device driver to fail the command in progress, if any, as well as all queued commands for the device with 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 should be 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. Note that for this command, the LUN in the *arg* parameter should be set by the SCSI device driver to the LUN ID of a LUN on this SCSI ID, which has been successfully started using the SCIOSTART operation.

  It is intended that the SCSI device driver only use this command when directed to do a *forced open*, which occurs in two possible situations. One is when it is desirable to force the device to drop a SCSI reservation. The other is when the device needs to be reset in order to clear an error condition (for example, when running diagnostics on this device).

  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) may have. If an SCIOSTART operation has not been previously issued, this command fails.

  The following values for the **errno** global variable should be 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.

A Typical 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, while 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 and any internal queuing that may be required occurs in this routine. The **dd_strategy** entry point then triggers the operation by calling the **dd_start** entry point. The **dd_start** routine merely 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 re-order 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, this 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. The SCSI adapter device driver also fills in 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 should be set to a value of zero. When a transaction completes, the **sc_intr** routine causes the **sc_buf** entry to be removed from the device queue and invokes the **iodone** kernel service, passing the just dequeued **sc_buf** structure for the device as the parameter. The **sc_start** routine is then re-invoked to process the next transaction on the device queue. The **iodone** kernel service then invokes the SCSI device driver **dd_iodone** entry point, signalling to the SCSI device driver that the particular transaction has completed.
5. 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 structure (or structures). It then calls the **iodone** kernel service with the original buffer pointer (or pointers) to notify the request's originator.

Other SCSI Design Considerations

The following points should be considered in the design of SCSI device and adapter device drivers:

- Responsibilities of the SCSI device driver
- SCSI options to the **openx** subroutine
- Using the SC_FORCED_OPEN option
- Using the SC_RETAIN_RESERVATION option
- Using the SC_DIAGNOSTIC option
- Closing the SCSI device
- SCSI error processing
- Length of data transfer for scsi commands
- Device driver and adapter device driver interfaces
- Performing SCSI dumps.

Responsibilities of the SCSI Device Driver

SCSI device drivers are responsible for the following actions:

- Interfacing with block I/O and logical volume device driver code in the AIX Base Operating System.
- Translating I/O requests from the AIX Base 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. In the AIX Base Operating System, it is assumed that other SCSI initiators may 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). 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 AIX Base Operating System must support some defined extended options in their open routine (that is, an **openx** subroutine). Additional extended options to the open are also allowed, but must not conflict with predefined open options. The defined extended options are bit flags in the *ext* open parameter. These options may be specified singly, or in combination with each other. The required *ext* options have the following values and are defined in the **<sys/scsi.h>** file:

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_RESV_04	Reserved for future expansion.
SC_RESV_05	Reserved for future expansion.
SC_RESV_06	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 first open. This forces the device to release another initiator's reservation. After the SCIORESET completes, other SCSI commands are sent as in a normal open. If any of the SCSI commands fail due to a reservation conflict, the open should be failed with the 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 since this request can force a device to drop a SCSI reservation. If the caller attempts to execute this system call without the proper authority, the SCSI device driver should return a value of –1, with the **errno** global variable set to EPERM.

Using the SC_RETAIN_RESERVATION Option

The SC_RETAIN_RESERVATION option causes the SCSI device driver to not 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. Note that for shared devices (for example, disk or CD-ROM), the SCSI device driver should OR together this option for all opens to a given device. Thus, if any caller requests this option, the close routine skips issuing 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 since this request can allow a program to monopolize a device (for example, if this is a non-shared device). If the caller attempts to execute this system call without the proper authority, the SCSI device driver should return a value of –1, with the **errno** global variable set to 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 in Diagnostic mode. It is intended that one or more **ioctl** operations 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 execute. If the caller attempts to execute this system call without the proper authority, the SCSI device driver should return a value of –1 with the **errno** global variable set to the value EPERM. The SC_DIAGNOSTIC option may only be executed 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 EACCES. Once successfully opened with the SC_DIAGNOSTIC flag, the SCSI device driver is placed in Diagnostic mode for the selected device.

The remaining options for the *ext* parameter are reserved for future requirements.

Implementation note: The following chart 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 cmds Close: no SCSI cmds
retain	Open: normal Close: no RELEASE
force	Open: normal, except SCIORESET issued prior to any SCSI commands
diag + retain	Open: no SCSI cmds Close: no SCSI cmds
diag + force	Open: issue SCIORESET otherwise, no SCSI cmds issued Close: no SCSI cmds

EXT OPTIONS	
openx *ext* option	Device Driver Action
force + retain	Open: normal, except SCIORESET issued prior to any SCSI commands Close: no RELEASE
diag + force +retain	Open: issue SCIORESET otherwise, no SCSI cmds issued Close: no SCSI cmds

Closing the SCSI Device

When a SCSI device driver is preparing to close a device through the SCSI adapter device driver, it must insure that all transactions are complete. When the SCSI adapter device driver receives an SCIOSTOP **ioctl** operation, if there are pending I/O requests, the **ioctl** operation does not return until they have all completed. New requests received during this time are rejected from the adapter device driver's **ddstrategy** routine.

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 knows nothing about particular SCSI commands, and is not responsible for device error recovery.

Length of Data Transfer for SCSI Commands

Commands initiated by the SCSI device driver internally or as subordinates to a transaction from above must have data phase transfers of 256 bytes or less to prevent DMA/CPU memory conflicts. A length of 256 or less indicates to the SCSI adapter device driver that data phase transfers are to be handled internally (in its address space).This is required to prevent DMA/CPU memory conflicts for the SCSI device driver. The SCSI adapter device driver specifically interprets a byte count of 256 or less as an indication that it may not perform data phase DMA transfers directly to or from the destination buffer.

The actual DMA transfer goes to a dummy buffer inside the SCSI adapter device driver and then is block-copied to the destination buffer. Internal SCSI device driver operations that typically have small data transfer phases are SCSI control-type commands such as mode select, mode sense, and request sense. However, the comments of this discussion apply to any command received by the SCSI adapter device driver that has a data phase size of 256 bytes or less.

Internal commands with data phases larger than 256 bytes require the SCSI device driver to specifically allocate the required memory on the process level. The memory pages containing this memory may not be accessed by the CPU (that is, the SCSI device driver) in any way from the time the transaction is passed to the SCSI adapter device driver until the SCSI device driver receives the **iodone** call for the transaction.

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 AIX 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 AIX 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_open**, **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.

SCSI adapter device driver writers should be aware that system services that provide 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** to be processed. Using this interface, a SCSI write command can be executed 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. No queuing of **sc_buf** structures is supported during dump processing, since the dump routine executes essentially as a subroutine call from the caller's dump routine. Control is returned when the entire **sc_buf** has been processed.

Also, both adapter device driver and device driver writers should be aware that any error occurring during the DUMPWRITE option is considered fatal. Therefore, no error recovery is employed during the DUMPWRITE. Return values from the call to the **dddump** routine indicate the failure.

Successful completion of the selected operation is indicated by a zero 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. Note that 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 the SCSI adapter device driver was passed an invalid request, such as attempting a DUMPSTART before successfully executing a DUMPINIT.

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 a status before the passed command timeout value expired.

Related Information

Logical Volume Subsystem on page 10–1.
Communication I/O Subsystem on page 8–1.
Kernel Environment Programming on page 1–6.
The Logical File System Kernel Kervices on page 6–12
Device Driver Concepts Overview on page 2–1.
Understanding Block I/O Device Drivers on page 3–1.

Appendix A. Alphabetical List of Kernel Services

This list provides the names of all kernel services and the execution environment from which each can be called. A kernel service can either be called in both the process and the interrupt environments, or only in the process environment.

ackque	Available in the process environment only.
add_arp_iftype	Available in both the process and interrupt environments.
add_domain_af	Available in both the process and interrupt environments.
add_input_type	Available in both the process and interrupt environments.
add_netisr	Available in both the process and interrupt environments.
add_netopt	Available in both the process and interrupt environments.
as_att	Available in the process environment only.
as_det	Available in the process environment only.
attchq	Available in the process environment only
audit_svcbcopy	Available in the process environment only.
audit_svcfinis	Available in the process environment only.
audit_svcstart	Available in the process environment only.
bawrite	Available in the process environment only.
bdwrite	Available in both the process and interrupt environments.
bflush	Available in the process environment only.
binval	Available in the process environment only.
blkflush	Available in the process environment only.
bread	Available in the process environment only.
breada	Available in the process environment only.
brelse	Available in both the process and interrupt environments.
bwrite	Available in the process environment only.
canclq	Available in the process environment only.
cfgnadd	Available in the process environment only.
cfgndel	Available in the process environment only.
clrbuf	Available in both the process and interrupt environments.
clrjmpx	Available in both the process and interrupt environments.
copyin	Available in the process environment only.
copyinstr	Available in the process environment only.
copyout	Available in the process environment only.

creatd	Available in the process environment only.
creatp	Available in the process environment only.
creatq	Available in the process environment only.
curtime	Available in both the process and interrupt environments.
d_align	Available in both the process and interrupt environments.
d_clear	Available in both the process and interrupt environments.
d_complete	Available in both the process and interrupt environments.
d_init	Available in both the process and interrupt environments.
d_mask	Available in both the process and interrupt environments.
d_master	Available in both the process and interrupt environments.
d_move	Available in both the process and interrupt environments.
d_roundup	Available in both the process and interrupt environments.
d_slave	Available in both the process and interrupt environments.
d_unmask	Available in both the process and interrupt environments.
del_arp_iftype	Available in both the process and interrupt environments.
del_domain_af	Available in both the process and interrupt environments.
del_input_type	Available in both the process and interrupt environments.
del_netisr	Available in both the process and interrupt environments.
del_netopt	Available in both the process and interrupt environments.
delay	Available in the process environment only.
deque	Available in the process environment only.
detchq	Available in the process environment only.
devdump	Available in both the process and interrupt environments.
devstrat	Available in both the process and interrupt environments.
devswadd	Available in the process environment only.
devswdel	Available in the process environment only.
devswqry	Available in both the process and interrupt environments.
dmp_add	Available in the process environment only.
dmp_del	Available in the process environment only.
dstryd	Available in the process environment only.
dstryq	Available in the process environment only.
DTOM macro	Available in both the process and interrupt environments.
e_post	Available in both the process and interrupt environments.

e_sleep	Available in the process environment only.
e_sleepl	Available in the process environment only.
e_wait	Available in the process environment only.
e_wakeup	Available in both the process and interrupt environments.
enque	Available in the process environment only.
errsave	Available in both the process and interrupt environments.
find_arp_iftype	Available in both the process and interrupt environments.
find_input_af	Available in both the process and interrupt environments.
find_input_type	Available in both the process and interrupt environments.
fp_access	Available in the process environment only.
fp_close	Available in the process environment only.
fp_fstat	Available in the process environment only.
fp_getdevno	Available in the process environment only.
fp_getf	Available in the process environment only.
fp_hold	Available in the process environment only.
fp_ioctl	Available in the process environment only.
fp_lseek	Available in the process environment only.
fp_open	Available in the process environment only.
fp_opendev	Available in the process environment only.
fp_poll	Available in the process environment only.
fp_read	Available in the process environment only.
fp_readv	Available in the process environment only.
fp_rwuio	Available in the process environment only.
fp_select	Available in the process environment only.
fp_write	Available in the process environment only.
fp_writev	Available in the process environment only.
fubyte	Available in the process environment only.
fuword	Available in the process environment only.
getadsp	Available in the process environment only.
getblk	Available in the process environment only.
getc	Available in both the process and interrupt environments.
getcb	Available in both the process and interrupt environments.
getcbp	Available in both the process and interrupt environments.

getcf	Available in both the process and interrupt environments.
getcx	Available in both the process and interrupt environments.
geteblk	Available in the process environment only.
geterror	Available in both the process and interrupt environments.
getexcept	Available in both the process and interrupt environments.
getpid	Available in both the process and interrupt environments.
getuerror	Available in the process environment only.
gfsadd	Available in the process environment only.
gfsdel	Available in the process environment only.
i_clear	Available in the process environment only.
i_disable	Available in both the process and interrupt environments.
i_enable	Available in both the process and interrupt environments.
i_init	Available in the process environment only.
i_mask	Available in both the process and interrupt environments.
i_reset	Available in both the process and interrupt environments.
i_sched	Available in both the process and interrupt environments.
i_unmask	Available in both the process and interrupt environments.
if_attach	Available in both the process and interrupt environments.
if_detach	Available in both the process and interrupt environments.
if_down	Available in both the process and interrupt environments.
if_nostat	Available in both the process and interrupt environments.
ifa_ifwithaddr	Available in both the process and interrupt environments.
ifa_ifwithdstaddr	Available in both the process and interrupt environments.
ifa_ifwithnet	Available in both the process and interrupt environments.
ifunit	Available in both the process and interrupt environments.
init_heap	Available in the process environment only.
initp	Available in the process environment only
.io_att	Available in both the process and interrupt environments.
io_det	Available in both the process and interrupt environments.
iodone	Available in both the process and interrupt environments.
iostadd	Available in the process environment only.
iostdel	Available in the process environment only
iowait	Available in the process environment only.

kmod_entrypt	Available in the process environment only
kmod_load	Available in the process environment only.
kmod_unload	Available in the process environment only.
kmsgctl	Available in the process environment only.
kmsgget	Available in the process environment only.
kmsgrcv	Available in the process environment only.
kmsgsnd	Available in the process environment only.
lockl	Available in the process environment only.
loifp	Available in both the process and interrupt environments.
longjmpx	Available in both the process and interrupt environments.
lookupvp	Available in the process environment only.
looutput	Available in both the process and interrupt environments.
m_adj	Available in both the process and interrupt environments.
m_cat	Available in both the process and interrupt environments.
m_clget	Available in both the process and interrupt environments.
m_clgetx	Available in both the process and interrupt environments.
m_collapse	Available in both the process and interrupt environments.
m_copy	Available in both the process and interrupt environments.
m_copydata	Available in both the process and interrupt environments.
m_dereg	Available in the process environment only.
m_free	Available in both the process and interrupt environments.
m_freem	Available in both the process and interrupt environments.
m_get	Available in both the process and interrupt environments.
m_getclr	Available in both the process and interrupt environments.
m_getclust	Available in both the process and interrupt environments.
M_HASCL macro	Available in both the process and interrupt environments.
m_pullup	Available in both the process and interrupt environments.
m_reg	Available in the process environment only.
MTOCL macro	Available in both the process and interrupt environments.
MTOD macro	Available in both the process and interrupt environments.
net_attach	Available in the process environment only.
net_detach	Available in the process environment only
net_error	Available in both the process and interrupt environments.

net_sleep	Available in the process environment only.
net_start	Available in the process environment only.
net_start_done	Available in both the process and interrupt environments.
net_wakeup	Available in both the process and interrupt environments.
net_xmit	Available in both the process and interrupt environments.
panic	Available in both the process and interrupt environments.
peekq	Available in both the process and interrupt environments
pfctlinput	Available in both the process and interrupt environments.
pffindproto	Available in both the process and interrupt environments.
pidsig	Available in both the process and interrupt environments.
pgsignal	Available in both the process and interrupt environments.
pin	Available in the process environment only.
pincf	Available in both the process and interrupt environments.
pincode	Available in the process environment only.
pinu	Available in the process environment only.
pio_assist	Available in both the process and interrupt environments.
prochadd	Available in the process environment only.
prochdel	Available in the process environment only.
purblk	Available in the process environment only.
putc	Available in both the process and interrupt environments.
putcb	Available in both the process and interrupt environments.
putcbp	Available in both the process and interrupt environments.
putcf	Available in both the process and interrupt environments.
putcfl	Available in both the process and interrupt environments
putcx	Available in both the process and interrupt environments.
qryds	Available in the process environment only.
queryd	Available in the process environment only.
queryi	Available in both the process and interrupt environments.
queryp	Available in the process environment only.
raw_input	Available in both the process and interrupt environments.
raw_usrreq	Available in both the process and interrupt environments.
readq	Available in the process environment only
rqc	Available in the process environment only

rqd	Available in the process environment only.
rqgetw	Available in the process environment only.
rqputw	Available in the process environment only.
rtalloc	Available in both the process and interrupt environments.
rtfree	Available in both the process and interrupt environments.
rtinit	Available in both the process and interrupt environments.
rtredirect	Available in both the process and interrupt environments.
rtrequest	Available in both the process and interrupt environments.
schednetisr	Available in both the process and interrupt environments.
selnotify	Available in both the process and interrupt environments.
setjmpx	Available in both the process and interrupt environments.
setpinit	Available in both the process and interrupt environments.
setuerror	Available in the process environment only.
sig_chk	Available in the process environment only.
sleep	Available in the process environment only.
subyte	Available in the process environment only.
suser	Available in the process environment only.
suword	Available in the process environment only.
talloc	Available in the process environment only.
tfree	Available in both the process and interrupt environments.
timeout	Available in both the process and interrupt environments.
timeoutcf	Available in the process environment only.
trcgenk	Available in both the process and interrupt environments.
trcgenkt	Available in both the process and interrupt environments.
tstart	Available in both the process and interrupt environments.
tstop	Available in both the process and interrupt environments.
uexadd	Available in the process environment only.
uexblock	Available in both the process and interrupt environments.
uexclear	Available in both the process and interrupt environments.
uexdel	Available in the process environment only.
uiomove	Available in the process environment only.
unlockl	Available in the process environment only.
unpin	Available in both the process and interrupt environments.

unpincode	Available in the process environment only.
unpinu	Available in both the process and interrupt environments.
untimeout	Available in both the process and interrupt environments.
uphysio	Available in the process environment only.
ureadc	Available in the process environment only.
uwritec	Available in the process environment only.
vec_clear	Available in the process environment only.
vec_init	Available in the process environment only.
vfsadd	Available in the process environment only.
vfsdel	Available in the process environment only.
vfsrele	Available in the process environment only.
vm_att	Available in both the process and interrupt environments.
vm_cflush	Available in the process environment only
vm_det	Available in both the process and interrupt environments
vm_handle	Available in the process environment only.
vm_makep	Available in the process environment only.
vm_mount	Available in the process environment only.
vm_move	Available in the process environment only.
vm_protectp	Available in the process environment only.
vm_qmodify	Available in the process environment only.
vm_release	Available in the process environment only.
vm_releasep	Available in the process environment only.
vm_umount	Available in the process environment only.
vm_write	Available in the process environment only.
vm_writep	Available in the process environment only.
vms_create	Available in the process environment only.
vms_delete	Available in the process environment only.
vms_iowait	Available in the process environment only.
vn_free	Available in the process environment only.
vn_get	Available in the process environment only
w_clear	Available in both the process and interrupt environments.
w_init	Available in both the process and interrupt environments.
w_start	Available in both the process and interrupt environments.

w_stop	Available in both the process and interrupt environments.
waitcfree	Available in the process environment only.
waitq	Available in the process environment only.
wakeup	Available in both the process and interrupt environments.
xmalloc	Available in the process environment only
xmattach	Available in the process environment only.
xmdetach	Available in both the process and interrupt environments.
xmemdma	Available in both the process and interrupt environments.
xmemin	Available in both the process and interrupt environments.
xmemout	Available in both the process and interrupt environments.
xmfree	Available in the process environment only.

Index

Reader's Comment Form

Kernel Extensions and Device Support Programming Concepts

SC23–2207–0

Please use this form only to identify publication errors or to request changes in publications. Your comments assist us in improving our publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page	Comments

Please contact your IBM representative or your IBM–approved remarketer to request additional publications.

Please print

Date ____________

Your Name ________________________________

Company Name ________________________________

Mailing Address ________________________________

____

____

Phone No. ( ) ____________________
Area Code

No postage necessary if mailed in the U.S.A

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 997
11400 Burnet Rd.
Austin, Texas 78758–3493

Fold Fold

Cut or Fold Along Line

Fold and Tape Please Do Not Staple Fold and Tape

SC23-2207-00