Bull DPX/20

Writing a Device Driver

AIX

ORDER REFERENCE
86 A2 29WG 04

Bull DPX/20

Writing a Device Driver

AIX

Software

November 1995

BULL S.A. CEDOC
Atelier de Reproduction
FRAN–231
331 Avenue Patton BP 428
49005 ANGERS CEDEX
FRANCE

ORDER REFERENCE
86 A2 29WG 04

Printed in France

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

Trademarks and Acknowledgements

We acknowledge the right of proprietors of trademarks mentioned in this book.

AIX® is a registered trademark of International Business Machines Corporation, and is being used under licence.

UNIX is a registered trademark in the USA and other countries licensed exclusively through X/Open.

Contents

About This Book

AIX Writing a Device Driver contains an overview of block and character device drivers and describes how to write a device driver for AIX Version 4.1. Also included is information on debugging. Refer to Chapter 20, “Packaging Software for Installation” in *AIX General Programming Concepts : Writing and Debugging Programs* for information on packaging device drivers.

Who Should Use This Book

This book is intended for programmers and software support personnel who need detailed information on writing device drivers. Readers of this book are expected to be familiar with the C programming language, AIX commands, subroutines, and special files.

How to Use This Book

Overview of Contents

Chapters 1 through 6 are intended for all readers of this book and discuss the following topics:

- Chapter 1 is a device driver overview.
- Chapter 2 discusses device input/output.
- Chapter 3 discusses interrupts.
- Chapter 4 is about memory management.
- Chapter 5 discusses synchronization and serialization.
- Chapter 6 is about device configuration methods.

Chapters 7 through 13 each discuss a particular type of device driver programming.

- Chapter 7 is about block device drivers.
- Chapter 8 is about SCSI device drivers.
- Chapter 9 discusses the IDE device drivers.
- Chapter 10 is about Virtual File Systems.
- Chapter 11 describes the STREAMS-based tty interface.
- Chapter 12 discusses implementing graphical input and 2D graphics device drivers.
- Chapter 13 discusses implementing a network device driver.
- Chapter 14 is about network interfaces and protocols.
- Chapter 15 contains information on debugging device drivers.
- Chapter 16 contains information on power management-aware 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 related to writing device drivers:

- *AIX Commands Reference*, Order Number 86 A2 73AP to 86 A2 78AP.
- *AIX General Programming Concepts : Writing and Debugging Programs*, Order Number 86 A2 65AP.
- *AIX Communications Programming Concepts*, Order Number 86 A2 70AP.
- *AIX Kernel Extensions and Device Support Programming Concepts*, Order Number 86 A2 71AP.
- *AIX Files Reference*, Order Number 86 A2 79AP.
- *AIX Problem Solving Guide and Reference*, Order Number 86 A2 56AP
- *AIX Technical Reference, Volume 5: Kernel and Subsystems*, Order Number 86 A2 85AP.
- *AIX Technical Reference, Volume 6: Kernel and Subsystems*, Order Number 86 A2 86AP.
- PowerPC *Architecture*.
- *Hardware Technical Information-General Architectures*, Order Number 86 A1 09WD.
- *UNIX System V Release 4, Programmer's Guide: STREAMS*. Englewood Cliffs, N.J.: Prentice-Hall, 1990.
- Angebranndt, Susan, Drewry, Raymond, Karlton, Philip, Newman, Todd, Packard, Keith and Scheifler, Robert W. *Strategies for Porting the X v11 Sample Server.* Massachusetts Institute of Technology. 1991.
- Fortune, Erik and Israel, Elias. *The X-Window Server*. Digital Press.
- Gettys, James, Newman, Ron and Scheifler, Robert W. *Xlib—C Language X Interface, MIT X Consortium Standard, X Version 11, Release 5.* MIT X Consortium 1991.
- Leffler, Samuel J., and others. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wessley. 1990.
- Patrick, Mark, and Sachs, George. *X11 Input Extension Library Specification. MIT X Consortium Standard. X Version 11, Release 5*. Hewlett-Packard Company, Ardent Computer, and the Massachusetts Institute of Technology. 1989, 1990, 1991.
- Patrick, Mark, and Sachs, George. *X11 Input Extension Protocol Specification. MIT X Consortium Standard. X Version 11, Release 5*. Hewlett-Packard Company, Ardent Computer, and the Massachusetts Institute of Technology. 1989, 1990, 1991.
- Sachs, George. *X11 Input Extension Porting Document. MIT X Consortium Standard. X Version 11, Release 5*. Hewlett-Packard Company and the Massachusetts Institute of Technology. 1989, 1990, 1991.
- Scheifler, Robert W. *X Window System Protocol, MIT X Consortium Standard, X Version 11, Release 5.* MIT X Consortium 1991.
- Womack, et al. *PEX Protocol Specification, Version 5.1, MIT X Consortium Standard*. Massachusettes Institute of Technology 1988, 1989, 1990, 1991, 1992.
- Womack, et al. *PEX Protocol Encoding Version 5.1, MIT X Consortium Standard*. Massachusettes Institute of Technology 1988, 1989, 1990, 1991, 1992.

Ordering Additional Copies of This Book

You can order publications from your sales representative or from your point of sale.

If you received a printed copy of *Documentation Overview* with your system, use that book for information on related publications and for instructions on ordering them.

To order additional copies of this book, use Order Number 86 A2 29WG.

Chapter 1. Device Driver Overview

Many computer programs are dedicated to working with attached devices in some way. For example, there are programs to send control characters to a printer, programs to receive characters from a terminal, and programs to read data from a tape. In a broad sense, each of these programs is a *device driver* because the program is dedicated to handling input from or output to a device. Such programs are usually regarded as being part of, or an extension of, the computer's operating system.

Any operating system that supports multitasking (such as AIX) needs some way to prevent one program from writing to, or changing the state of, some device that is already being accessed by another program. So, a multitasking operating system relies on the computer's processors to distinguish between privileged and non-privileged execution of instructions. Therefore, one must distinguish between programs that execute in privileged mode (kernel mode) and those that execute in user mode. The AIX kernel consists of all software that executes in kernel mode.

Even though AIX programs that execute in user mode can drive devices, such as a printer or some device attached to a serial port, they can only do so by invoking software that is part of the kernel. Because kernel device drivers are considerably more complex than drivers that execute in user mode, from here on, the term *device driver* will only refer to software that handles a device while executing in kernel mode.

Device drivers are more complex than user software for several reasons:

- Device drivers output data to a device or demand data from a device.

  This means that the driver may have to read or write to registers on a card attached to an I/O bus, or the driver may have to set up the means for the data to be transferred in some other way. A device driver is intimately interconnected with processor memory design, how the processor performs I/O, and with the architecture of the I/O bus attached to the system. So, device drivers are not portable; migrating the driver routines from one system to another often requires the routines to be rewritten.

- Device drivers may have to process interrupts generated by a card attached to the system I/O bus.

  When a terminal sends a character to the computer, or a printer runs out of paper, or a tape drive has completed writing a block of data, the card serving as an adapter between the device and the I/O bus on the computer generates an interrupt. The software routines, within a device driver, that process interrupts (called *interrupt handlers*) take some sort of action like buffering incoming data, or signaling a process. Because such interrupts occur *asynchronously*, meaning that they occur without regard to what instructions the computer's processors are executing, the interrupt may occur while a processor on the computer is in the middle of handling another interrupt. Therefore, interrupt handlers must be reentrant; in other words, they must be able to access shared resources (such as non-private data) and exclude concurrent access by any software including another instance of itself.

  Device driver routines that (asynchronously) execute in the context of handling an interrupt are said to be, *on the interrupt side*, and are occasionally referred to as the *device handler*, but this is not the same thing as a *network device handler*.

  Device driver routines that (synchronously) execute in the context of a calling process are said to be *on the call side*. For more information on concurrent access of shared data, see "Synchronization and Serialization" on page 5-1.

- Device drivers may have to execute *in real time*.

  Device drivers may have to respond to an interrupt, or perform some other function within a certain fixed period of time.

- A device driver is a collection of routines. There is no *main* routine.

  The routines are usually written in C and compiled to produce one or two Extended Object File Format (XCOFF) object files. The object files are linked to enable the kernel loader to resolve kernel symbols. As a result of linking, the loader section is filled out with a list of symbols to import from the kernel. The symbols are in the file **/lib/kernex.exp**. The linking also establishes the driver's configuration routine as the default entry point for beginning execution. A simple example is shown in "Sample Device Driver" on page 1-15.

Aspects of the Kernel that Affect Device Drivers

There are a number of attributes of the AIX kernel that affect device drivers:

- Kernel routines can only call kernel services.

  Kernel routines cannot call routines meant to execute in user mode. So, device driver routines are not linked with the C library **libc.a**, nor can they invoke system calls. There are some kernel DMA and timer routines in **libsys.a**, and there are some C library calls written to execute in kernel mode in the library **libcsys.a**. For more information on them, please refer to "Understanding Kernel Extension Binding," in *AIX Kernel Extensions and Device Support Programming Concepts*.

- Kernel code and data that is not pinned (explicitly or implicitly) is paged into system RAM from a paging logical volume on disk.

  So, one must distinguish between device driver routines that are to be collectively pinned, called *the bottom half*, and those that are to be paged, called *the top half*. Because device driver routines on the interrupt side must be pinned, the phrases *in the bottom half* and *on the interrupt side* are sometimes used as if they are synonyms, but they really are not synonyms. Due to real-time concerns, sometimes routines on the call side are placed in the bottom half.

- Kernel routines are difficult to debug.

  There is a kernel debugger, but it is not as easy to use as is **dbx**. References to improper addresses may cause data corruption (because the kernel is privileged) or a system crash. It is possible to trace the execution of a device driver with **printf**; but **printf** only prints to a native serial port, cannot be used in an interrupt handler, and may affect a device driver's timing.

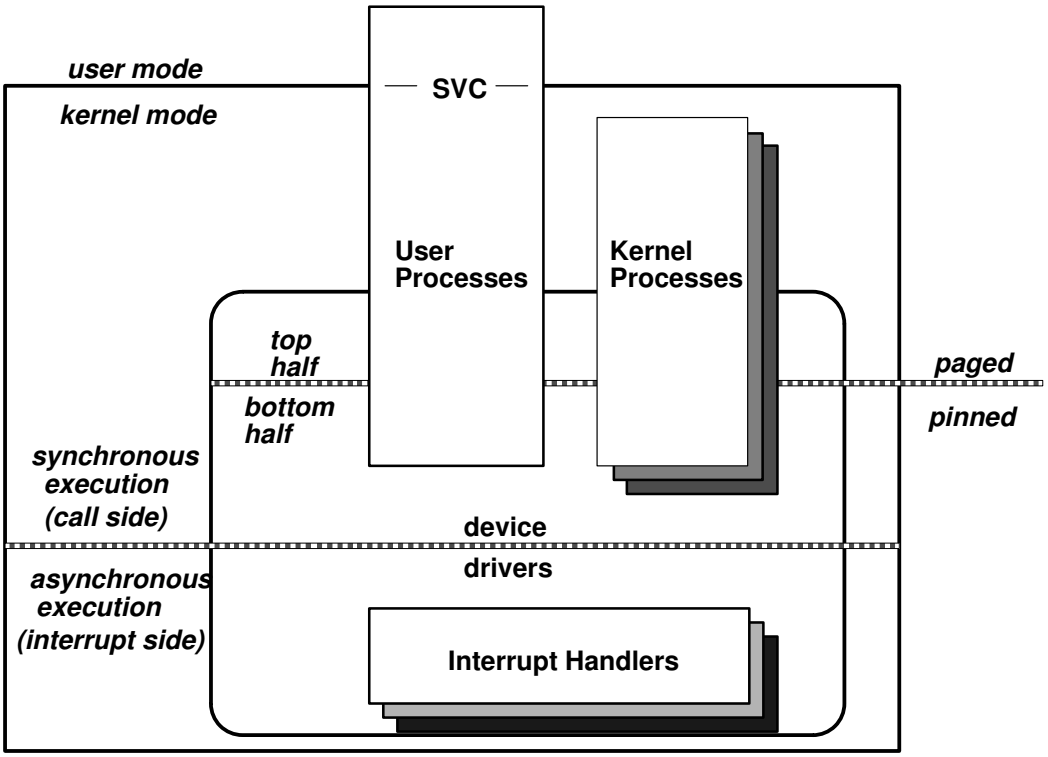
- Execution of kernel routines, in the context of a process, can be preempted by the scheduler in favor of a process with greater priority.

  This means that device driver routines cannot depend on disabling interrupts as being sufficient to avert concurrent access to shared resources. It also means that drivers (not just interrupt handlers) must be reentrant.

- The AIX kernel is dynamically extendible.

  Object files acceptable to the kernel loader, are bound into the AIX kernel while the computer is still operating; there is no need to restart the system. A device driver's routines are typically linked to form two object files (one for the top half, the other for the bottom half), that can be loaded or unloaded by a user program invoking the kernel loader with the **sysconfig** system call. Loading (*configuring*) files into the kernel is *extending the kernel*. A kernel extension, such as a device driver, is configured into the kernel while starting the system, or while the system is operating.

The AIX Kernel figure summarizes some aspects of the AIX kernel that affect device drivers. Note that AIX enables *kernel processes*, processes that execute entirely in kernel mode.



AIX Kernel

How Device Drivers Are Accessed

In many operating systems, a user who wants to transfer data to a device must execute a command specific to that device. In such an operating system, writing to tape requires a different command than writing to a terminal, or writing to a file.

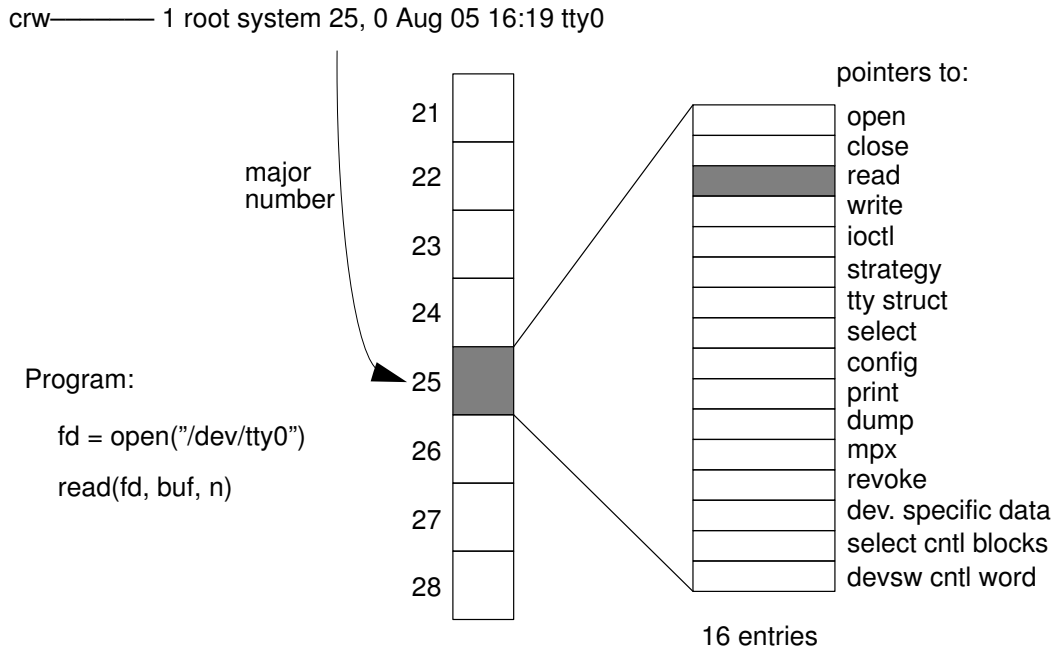
A feature of any UNIX operating system, such as AIX, is that I/O to a device is made to look like I/O to a file in the system's directory tree. A device is made ready for I/O by having its corresponding file (usually placed in the directory /**dev**) opened, and data is read from the device, or written to the device, by invoking **read** and **write** system calls on the corresponding file. The device (for example, a printer) is freed for access by other software by closing the file (called a *device special file*) associated with the device.

UNIX avoids the need to pass a device-specific parameter to system calls so that UNIX can present the same device interface to any user program accessing different devices. It does this by keeping device-specific data in the inode of the device special file. Such data includes:

- A flag marking the file as *special*.
- A major number identifying which device driver is to be invoked (such as for a tape drive or printer).
- A minor number identifying a particular device among the several devices handled by the device driver associated with the major number (for example, selects tape1 or printer3).
- A flag marking the device type as *character* or *block.*

The file, created by the **mknod** system call, is marked *special* so that system calls know to access a device, and not a file on disk.

The major number serves as an index into an array of structures. Each structure contains pointers to functions to be invoked when opening, closing, reading, writing, or performing whatever device operation the program requires. The array of these structures is called a *device switch table.* The Device Switch Table figure illustrates these structures.



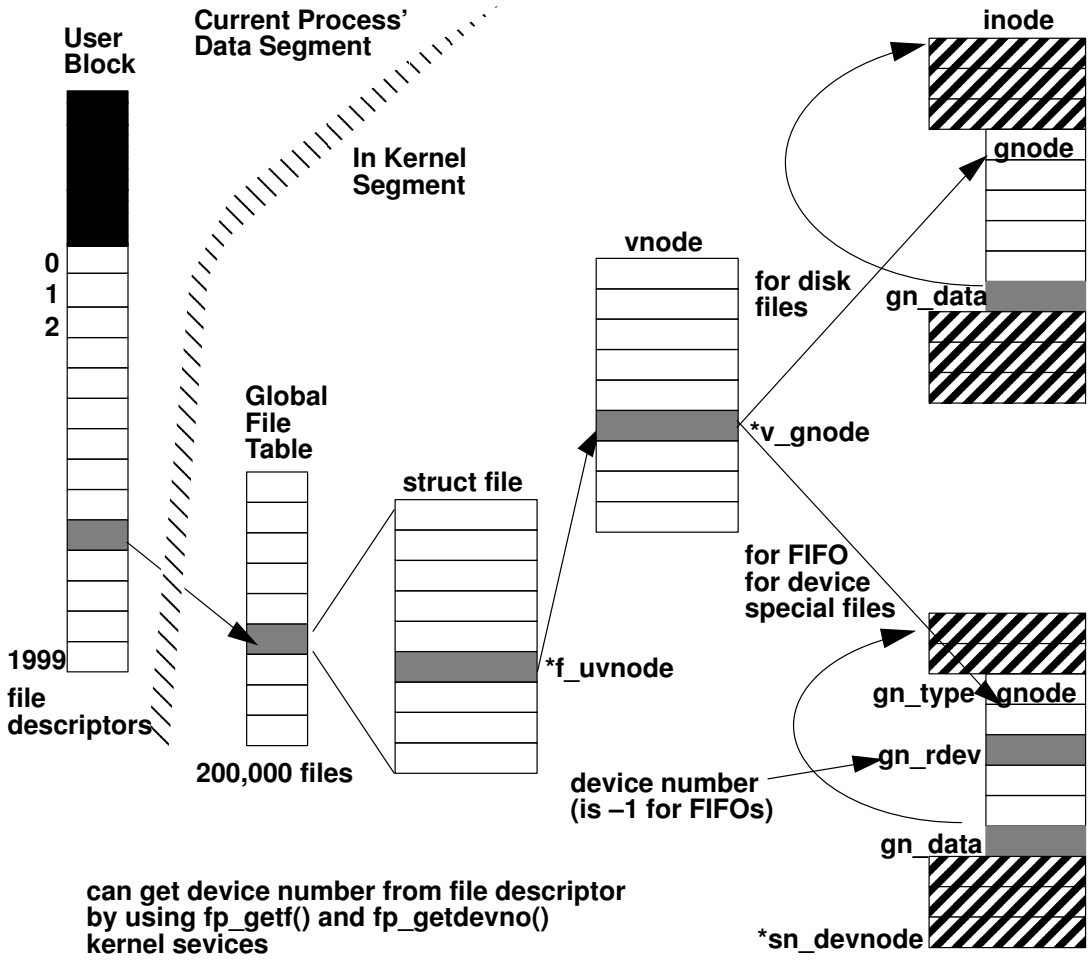
Device Switch Table

In AIX, the device switch table can have up to 256 such structures. When the driver is configured into the kernel, each function pointer in the structure is assigned the virtual address of the first instruction of an associated routine, that is, the virtual address of a routine's entry point. If a device driver does not define a particular routine (no drivers define all of them), use either a pointer to the function **nodev**, which returns ENODEV (in **sys**/**errno.h**), or a pointer to the function **nulldev**, which returns NULL.

In effect, the major number specifies which device driver to invoke. If there is a serial port with one driver, and, say, a multiport serial adapter which requires another driver, then each would have their own unique major number. On the other hand, if you have a high-density tape drive attached to one adapter, and a different, low-density tape drive attached to another adapter, but a driver supports both kinds of adapters, then there is only one major number needed to access either tape drive.

The minor number is used to distinguish between devices supported by the same driver. It typically serves as an index into an array, maintained by the driver, of structures containing device-specific information. For example, a terminal driver would need to keep track of the various baud rates or parity settings of each terminal.

Once a program has opened a file, it uses the file descriptor to determine the *device number* which combines the device's major and minor number into one integer. The program that configures the device driver into the kernel allocates a device number that is unique for the system. The figure From File Descriptor to Device Number shows the data structures involved in determining a device number from a file descriptor associated with the device's special file.



From File Descriptor to Device Number

On some UNIX systems, there is a device switch table for character drivers, and one for block drivers, hence the need for a flag **c** or **b** to mark the distinction. Some UNIX systems access a stream head (a generic interface to STREAMS modules or drivers) through a third device switch table. However, in AIX, the entry points to character drivers, block drivers, and stream heads are all invoked through the same device switch table. The kinds of routines that are expected to be included as entry points for a device driver vary with the type of device driver needed.

Types of Device Drivers

There are three types of device drivers in AIX:

- A block device driver supports a device that reads and writes data buffers of a (large) fixed size.
- A STREAMS device driver has some routines that are either invoked from a stream head, or from a STREAMS module, instead of from the device switch table.
- A character device driver is any driver that is not either of the other two types.

Block Device Drivers

Devices usually supported by a block device driver include: hard disk drives, diskette drives, CD-ROM readers, and tape drives. Block device drivers often provide two ways to access a block device:

raw access The buffer supplied by the user program is to be pinned in RAM as is.

block access The buffer supplied by the user program is to be copied to, or read from buffers in the kernel.

If the block device is accessed as *raw*, the driver can copy data from the pinned buffer to the device. In this case, the size of the buffer supplied by the user must be equal to, or some multiple of, the device's block size. The special file's name is usually prefixed by the letter **r** so that a user can tell which access type the block device has. For example, the name of a diskette drive's raw block special file is **rfd0**, and a special file name for a tape drive is **rmt0**.

Sometimes the term *character mode access* is used to mean raw access.

Otherwise, the block device is accessed as *block*. In this case, a **write** system call to such a device returns once the user buffer is copied to buffers in the kernel segment. The **write** call is asynchronous since the data in the kernel buffer is written out to the block device sometime after the **write** call returns. The size of the user buffer need not be a multiple of the device block size.

The term *buffer cache* refers to a collection of kernel buffers that are manipulated by some kernel services specifically associated with block devices. Although UNIX block device drivers have traditionally made use of the buffer cache, it is rarely used in AIX because buffering is more frequently done by memory mapping regions of the kernel segment with frames in RAM.

A block device driver with a block access type method is a driver that also provides a *strategy* routine to arrange accesses to device blocks so that overall access time is minimized. The strategy entry point is not invoked from a user program; rather, the entry point, which is in the device switch table, can be invoked by either of the following:

- Off-level interrupt handlers responsible for writing the buffer cache out to disk.
- The AIX Virtual Memory Manager to perform paging, that is, to page space for working segments, to disk files for memory-mapped files.

The block driver's strategy routine is intended for reading and writing buffers that are not necessarily contiguous on the device itself.

A tape device driver has a raw access method, but has no block access method. There is no use of kernel buffers, and there is no reason to provide a strategy entry point since tape does not lend itself to efficient random access.

For more information on implementing a block device driver, "Block Device Drivers" on page 7-1.

Block devices are often intended to contain a file system. A block device driver that interfaces to the Logical Volume Manager (LVM) enables the block device to support a journaled file system (JFS). For more information on this, see "Logical Volume Programming" in *AIX General Programming Concepts : Writing and Debugging Programs* and "Understanding Physical Volumes and the Logical Volume Device Driver" in *AIX Kernel Extensions and Device Support Programming Concepts*.

For a block device to contain a file system that is not provided with the operating system, kernel routines that interface between the virtual file system (VFS) and the block device driver must be provided. For more information on this, see "Virtual File Systems," in *AIX Kernel Extensions and Device Support Programming Concepts*, and "Writing a Virtual File System" on page 10-1.

STREAMS Device Drivers

Devices that may be supported by a STREAMS driver include: any device connected to the serial port (such as a terminal), or any device attached to a LAN or WAN (such as an Ethernet adapter).

Such devices lend themselves to support from STREAMS drivers because the STREAMS facility is flexible and modular. These qualities are well suited to implementing communication protocols.

Since the TTY subsystem in AIX Version 4.1 consists of STREAMS modules, if you want to support terminal processing from a serial adapter you must provide a STREAMS driver. For more information on this, see “STREAMS-Based TTY Subsystem Interface” on page 11-1. “Implementing Graphical Input and 2-D Graphics Device Drivers” on page 12-1 contains related information about the low-function terminal (LFT) subsystem that supports use of a console display.

AIX provides a STREAMS driver, the Data Link Protocol Interface (DLPI), which supports some LAN adapters. For more information on this, see “Implementing a Network Device Driver” on page 13-1.

A stream is a linked list of kernel modules, and consists of a stream head at one end of the list and a STREAMS device driver at the other. To visualize how a stream works, see the STREAMS Driver Entry Points figure on page 1-14.

The stream head (supplied with the operating system as part of STREAMS, a device driver writer does not need to write a stream head) contains some routines that are invoked from the device switch table, so the stream head is associated with a device special file in the AIX file tree.

A STREAMS driver has some routines that are either invoked by the stream head, or by a STREAMS module that had been inserted into the stream between the stream head and the STREAMS driver. The driver may, or may not, have any routines that are invoked from the device switch table.

For more information on this, see “STREAMS Overview,” in *AIX Communications Programming Concepts* and *UNIX System V, Release 4, Programmer’s Guide: STREAMS*.

Character Device Drivers

Devices that are supported by a character device driver include any device that reads or writes data a character at a time (such as printers, sound boards, or terminals). Also, any driver that has no associated hardware device (called a *pseudo-driver*) is treated as a character device driver. For example, /**dev**/**mem**, /**dev**/**kmem**, and /**dev**/**bus0** are *character pseudo-drivers*.

Graphics input devices and graphics capable displays are often supported by character device drivers. For more information on how to implement such drivers, see “Implementing Graphical Input and 2D Graphics Device Drivers” on page 12-1.

A character special file that has the mode flag S_ISVTX (also called the *sticky bit* because it causes the text of an executable to remain in memory after use) set, is a multiplexed character file. Special files are created and deleted in **/dev** as needed to support multiple ports connected to that adapter. For example, a serial adapter that supports multiple terminals would need to be a multiplexed character file. A multiplexed device driver contains an additional `xyzmpx` entry point. A pseudo-TTY (PTY) is an example of a multiplexed character device.

Access to multiplexed character device drivers is similar to that of 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.

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.

Device Driver Configuration

When a version of a device driver is written, to test the driver you need to create and load the driver's object files into the kernel. Sample code in this section, shown in several parts, shows the basic steps of compiling, linking, loading, and testing a pseudo-driver.

Assume that the device special file is `/dev/xyz`. The device driver's object files are usually kept in the directory /**usr**/**lib**/**drivers**, but for this simple example, the object file `xyz` is kept in the current directory. You can give a driver's object file any name permitted within an AIX file system.

To configure a device driver object file `./xyx` into the AIX kernel, a user program with root user authority, here written in C, extends the kernel:

```
struct cfg_load cfg;
cfg.path = "./xyz";
sysconfig(SYS_KLOAD, &cfg, sizeof(cfg)):
```

Continuing the sample code, the configuring program needs to pick major and minor numbers that are not already in use:

```
majorno = 99;       /* To avoid ODM for now */
minorno = 0;
device_number = makedev(majorno, minorno); /* see sysmacros.h */
```

A program that configures a driver into the kernel does not really just guess which major and minor numbers to use. Associated with the Object Data Manager (ODM) are user routines (such as the **genmajor** and **genminor** subroutines) for determining what numbers to use and for allocating the numbers. Use of ODM is described later, but major number 99 is picked in the example just as an illustration. Calling **makedev** combines the major number and the minor number into one integer, the device number.

A configuring program usually creates the device special file to be associated with the device:

```
mknod ("/dev/xyz", 0666 | _S_IFCHR, device_number);
```

Now, the configuring program invokes the device driver's configure entry point:

```
struct cfg_dd xyzcfg;
xyzcfg.kmid = cfg.kmid; /* kernel module ID from sysconfig */
xyzcfg.devno = device_number;
xyzcfg.cmd = CFG_INIT;
sysconfig(SYS_CFGDD, &xyzcfg, sizeof(xyzcfg));
```

Control now passes to the default entry point for the device driver module. When linking the device driver routines object file, the entry point specified should be the symbolic name of the configuration entry point, in this case `xyzconfig`. A more complete example is shown in the configuration program in "Sample Device Driver" on page 1-15.

STREAMS device drivers are configured into the kernel by invoking the **strload** and **str_install** commands after editing the /**etc**/**pse.cfg** file. The **strload** command extends the kernel by loading the Portable STREAMS Environment (pse) kernel extension. The **str_install** command extends the kernel by issuing a series of calls to **sysconfig** as indicated by entries in the file /**etc**/**pse.cfg**.

Object Data Manager (ODM) Database

Many UNIX systems have ASCII stanza files that are edited as part of configuring a driver into the kernel. For example, in some UNIX systems you add stanzas to a file such as **/etc/system** or **/etc/master**. In AIX, such stanza files are kept in a directory, **/etc/objrepos**, called the ODM database. In AIX, you do not directly modify the files in the ODM database, but instead you call ODM routines, or execute ODM commands to modify the ODM database. The environment variable **ODMDIR** specifies which directory the ODM routines reference.

The files in an ODM database are indexed ASCII files called ODM object classes. Object classes can be thought of as tables where each row is an object and each column is a field within each object. For example, in the file **CuAt**, there is an object with name `tok0` that has attribute `dma_lvl` and value `0x5`. This object says the Token Ring card associated with `/dev/tok0` has DMA level 5.

The following object classes are significant for device drivers:

PdDv	Predefined (supported) devices. (Not necessarily actually installed on the system)
PdAt	Predefined attributes of the predefined devices
PdCn	Predefined connections/dependencies
CuDv	Customized (defined and/or available) devices
CuAt	Customized attributes of system and customized devices
CuDep	Customized dependencies, which devices/subsystems require which others
CuDvDr	Customized device driver resources. For example, ensures unique major numbers
CuVPD	Customized vital product data (for Micro Channel adapters)
Config_Rules	List of configuration methods for **cfgmgr** command to execute product inventory LPP history

A device method is an executable program, usually written in C, that modifies an ODM object class associated with a device. A device method is invoked by a user with root user authority, or when the command **cfgmgr** is called by rc.boot in RAM disk when the computer system is started.

The following types of device methods should be provided for a device driver. These methods are usually kept in the directory **/usr/lib/methods**.

- Define method (causes device to be defined).

  A define method's main task is to retrieve device data from PdDv in ODM and create a CuDv object. Also, it ensures that a parent device exists in the CuDv object.

- Configure method (causes device to be available).

  A configure method should perform the following steps:

  a. Display LED value on system LED panel.

  b. Verify that a parent device is available (in ODM).

  c. Verify that a device is present.

  d. Invoke the **busresolve** system call to get an interrupt level assigned to the device.

  e. Extend the kernel by calling **sysconfig**.

  f. Generate major and minor numbers and create a special file entry in **/dev**.

  g. Build a device dependent structure (DDS).

h. Invoke the device driver's config entry point by calling **sysconfig** and passing it the DDS.

i. Downloading any microcode needed by the adapter.

j. Updating CuDv with Vital Product Data and making the device available.

In addition to the define method and the configure method, other methods (for example, undefine, unconfigure, start, stop, and change methods) may also be needed.

For more information on configuration methods, see "Device Configuration Methods" on page 6-1 or "Object Data Manager (ODM)" in *AIX General Programming Concepts : Writing and Debugging Programs*.

Device Driver Entry Points

We now consider how control passes from a user program to a device driver entry point associated with the system call that the user program invoked. As shown in the Device Switch Table figure, on page 1-4, there are up to eleven entry points listed in the device switch table for a particular driver: open, close, read, write, ioctl, strategy, select, config, dump, mpx, and revoke. The entry point, "print," is not used. Even though the strategy and dump driver routines have entry points in the device switch table, they cannot be invoked by a system call in a user program.

Those routines which can be invoked from a system call in a user program, whose entry points are listed in the device switch table, are collectively known as the *device head*, and are said to perform the *device head role* within the driver. These routines are expected to return control to the user program that invoked them once their task is complete. Even though these routines are placed in the same object module, they rarely interact with each other (for example, they don't call each other). These routines merely share resources such as kernel data structures and the device itself. Like any program, a device driver can define other routines as needed, routines that may be invoked by any other routine in the driver.

For information on how to write each routine associated with an entry point in the device switch table, please refer to a complete list of such routines in Chapter 2, "Device Driver Operations" in *AIX Technical Reference, Volume 5: Kernel and Subsystems*. The following discussion focuses on routines common to most drivers.

Note that a routine's entry point is customarily labeled by prefixing a routine's function with some device-specific abbreviation based on the device special file name. For example, the entry point associated with a routine that opens a terminal device is labeled, **ttyopen**, and one that closes the device is, **ttyclose**. But, one is free to label entry points with any symbol that a compiler and linker will accept.

xyzconfig Entry Point

A device driver's configuration entry point is called when a program directs the kernel loader to configure the device driver's object file into the kernel. The configuration entry point is also called when the device driver is being removed from the kernel or when certain data is queried from the device.

Below are the commands that the device driver's configure method passes as a parameter to the **sysconfig** system call:

- CFG_INIT

  In this case, the tasks that the configuration routine might perform are such things as, placing entry points to other routines into the switch table by invoking **devswadd**, or initializing and allocating kernel data structures associated with the device driver, or initializing the adapter, or downloading microcode to the card attached to the device.

  Also, the **sysconfig** system call usually passes a Device Dependent Structure (DDS) to the configuration routine when initializing the device. One defines the DDS in whatever fashion is necessary for the driver. For example, a serial device driver might require the

DDS to contain initial values for baud rate, bits per character, parity bit settings, and so on.

- CFG_TERM

  In this case, the routine checks for any outstanding open file descriptors and releases any associated resources.

- CFG_QVPD

  In this case, the routine returns Vital Product Data from the card attached to the device.

xyzopen and xyzclose Entry Points

These routines usually perform the following functions:

- Allocate or free resources for this device instance.

  This is often where data structures are allocated or freed from the kernel heap.

  This is often where the bottom half is pinned or unpinned, and this is where interrupt handlers are often registered for use by the kernel, or removed from use by the kernel.

- Update use counts and semaphores if exclusive access required.

The **xyzopen** routine is invoked by **open** or **creat** system calls issued from a user program, or is invoked from an **fp_open** or **fp_opendev** kernel service call issued from a kernel extension.

The **xyzclose** routine is invoked by the **close** system call issued from a user program, or is invoked from the **fp_close** kernel service call issued from a kernel extension.

xyzread Entry Point

This routine usually does the following:

- Returns a buffer of whatever data was collected from the device (via the device driver's interrupt handler).

  A call to the **read** system call from a character device amounts to transferring data from a kernel buffer that had been populated by this device's interrupt handler to a buffer supplied by the user (which is usually outside that user's data segment)

- From a block device, initiates block I/O requests via the **uphysio** kernel service.

  This is referred to as *raw access* since no kernel buffers are being used for reading.

- From a streams device, causes the stream head or module to invoke the device driver's **xyzput** entry point.

Depending on how the device special file was opened, the **xyzread** routine usually puts the calling process to sleep until the data requested is available.

The **xyzread** routine is invoked by the **read** system call issued from a user program, or is invoked by the **fp_read** kernel service call issued from a kernel extension.

xyzwrite Entry Point

This routine usually does the following:

- Outputs data to a block device.

  An explicit write to a block device, one not using the **xyzstrategy** routine, is raw access. This initiates block I/O requests via the **uphysio** kernel service

- Outputs data to a character device.

  A write to a character device transfers data from a buffer supplied by the user, which is usually out of that user's data segment, pointed to by the **uio** structure, to any buffer needed by the device adapter, usually one character at a time.

- Outputs data to a STREAMS device.

  A write to a STREAMS device causes the stream head (or some module in the Stream) to invoke the streams driver's **xyzwput** routine.

The **xyzwrite** routine is invoked by the **write** system call issued from a user program, or is invoked by the **fp_write** kernel service routine issued from a kernel extension.

xyzstrategy Entry Point

This routine usually schedules read or write requests of a block device. Such requests are added to a queue of pending I/O requests for the device. The queue can be sorted to optimize device access; for example, one may wish to have disk blocks organized so that any that are within the same cylinder (under the drive's read/write head) are input or output in one operation.

The buffer supplied to the **xyzstrategy** routine must be pinned in RAM, because the actual I/O with the buffer is asynchronous (it may happen after the routine exits).

This routine calls the **iodone** kernel service once it's finished.

The **xyzstrategy** routine is invoked indirectly by the following:

- The **uphysio** kernel service
- The Logical Volume Manager (LVM)
- The Virtual Memory Manager (VMM) (for handling page faults)

For more information about use of strategy routines for block devices, see "Block Device Drivers" on page 7-1.

xyzioctl Entry Point

This routine usually performs functions that are not done by any of the other routines mentioned up to now. Usually, the **xyzioctl** routine modifies, or inspects the state of the attached device, and reports any results back to the calling program. For example, a terminal driver would have its **ttyioctl** routine enable the calling program to modify baud rate, bits per character, and so on.

This routine need not be synchronous since a device may not permit immediate action. For example, a program that calls ioctl(**CIOSTART**) on a LAN adapter requires a subsequent ioctl(**CIOGETSTAT**) to determine whether the first **ioctl** call is complete.

Not every driver has this entry point, but if this routine is present, it must support the **IOCINFO** command option which tells **xyzioctl** to return a structure that describes the attached device.

The **xyzioctl** routine is invoked by the **ioctl** system call issued from a user program, or is invoked by the **fp_ioctl** kernel service call issued by a kernel extension.

xyzmpx Entry Point

Character device drivers can 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 **ddmpx** routine, passing 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 **ddopen** routine with the channel ID received from the **ddmpx** routine to allow for any special processing (such as initializing a device or allocating a resource). 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 **ddclose** routine is called once for every close that had an associated open or create request. Once the file system determines that the last close has been issued for a channel, the multiplexed driver **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 calls to **fork** and **dup** subroutines) cannot. Because the last close for a channel can be recognized by the channel deallocation call to the **ddmpx** routine, keeping a count is not always required.

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 can 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.

xyzselect Entry Point

This routine enables notifying a calling thread about multiple I/O events.

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 (the **POLLIN** flag), device available for writing (the **POLLOUT** flag), and exceptional condition outstanding (the **POLLPRI** flag).

The select routine should check the current state of the device and set the corresponding 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 store in memory 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 other operating system's 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 had 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.

While 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. You can 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.

The **xyzselect** routine is invoked by the **select** or **poll** system calls issued from a user program, or is invoked by the **fp_select** kernel service call issued by a kernel extension.

xyzrevoke Entry Point

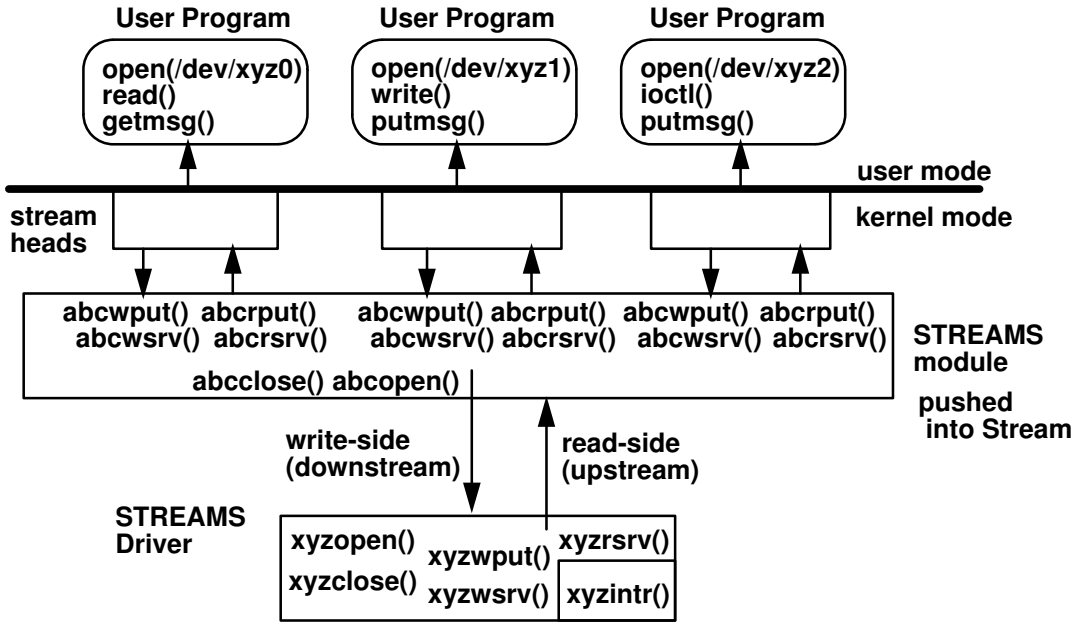
Only device drivers in the Trusted Computing Path must provide a **ddrevoke** routine so that the calling thread can terminate all other threads that have the device special file open and are currently in the wait state.

xyzdump Entry Point

Device drivers provide the **dddump** entry point when their respective devices can be selected as an output device for system dump data.

STREAMS Entry Points

A STREAMS driver may include routines invoked directly from the device switch table. However, a STREAMS driver usually handles all processing by using a write-side (downstream) put routine, and one or two optional service routines. Like any device driver, a STREAMS driver also has an interrupt handler which performs functions similar to those performed by a read-side (upstream) put routine of a STREAMS module. These relationships are shown in the STREAMS Driver Entry Points figure.



STREAMS Driver Entry Points

For more information on how to write a STREAMS module or device driver, see *UNIX SYSTEM V, Release 4, Programmer's Guide: STREAMS*.

xyzwput Entry Point

The write-side put routine receives messages from the stream head or STREAMS module upstream. The stream head converts **write** and **ioctl** system calls, issued from the user program, into messages, and then sends the messages downstream by invoking the write-side put routine of whatever STREAMS module or driver is downstream. The stream head handles the **read** system call issued from the program; the STREAMS driver does not.

This routine is invoked by the stream head or STREAMS module upstream from the driver.

A STREAMS driver does not have a read-side put routine. The interrupt handler assumes the role of receiving data from the device's adapter.

xyzwsrv, xyzrsrv Entry Points

A STREAMS driver can optionally support either a write-side or read-side service routine, neither, or both.

When a driver's write-side put routine determines that it must defer writing data to the device (flow control), it must place the data on a write-side queue. A kernel process, which is part of the Portable STREAMS Environment (PSE), eventually invokes the driver's write-side service routine, which checks the write-side queue and attempts to write the data to the device, or enqueues the data on the write-side queue so it can make another attempt later.

Similarly, the device driver's interrupt handler may place a buffer of data coming in from the device on a read-side queue. The PSE kernel process eventually invokes the driver's read-side service routine, which checks the read-side queue and then processes the data. Once the processing is complete, the read-side service routine invokes the read-side put routine of the STREAMS module upstream.

The data to be handled by a service routine is placed on a queue, as follows:

- The write-side put routine must call **putq** to place messages on a write-side service queue, so that invoking the **xyzwsrv** routine later will process that message.
- The interrupt handler must call **putq** to place messages on a read-side service queue for deferred processing by the **xyzrsrv** routine.
- A service routine gets messages off its own queue by calling **getq**.

Service routines are invoked by the STREAMS scheduler, implemented within a kernel process in AIX. The STREAMS scheduler is part of the **PSE** kernel extension.

Sample Device Driver

For greatest simplicity, consider a pseudo-driver (one having no associated device), whose routines are minimal and yet demonstrate an outline of what a device head looks like. As with any sample code in this book, the following warning applies.

Warning: The source code examples provided are only intended to assist in the development of a working software program. The source code examples may not function as written: additional code is required. In addition, the source code examples may not compile and may not bind successfully as written. The source code examples are provided, both individually and as one or more groups, "AS IS" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the source code examples, both individually and as one or more groups, is with you. Should any part of the source code examples prove defective, you assume the entire cost of all necessary servicing, repair, or correction. The contents of the source code examples are not warranted, individually or as one or more groups, to meet your requirements or to be error-free. Improvements and/or changes in the source code examples may be made at any time. Changes may be made periodically to the information in the source code examples;

these changes may be reported, for the sample device drivers included herein, in new editions of the examples. References in the source code examples to products, programs, or services shall not be viewed as an endorsement of any kind. A reference to a product must not be construed to imply that the product is available or will be made available in your country. Any reference to a licensed program in the source code examples is not intended to state or imply that only the particular licensed program can be used. Any functionally equivalent program can be used.

Files for Sample XYZ Device Driver

The sample device driver has the following files located in a user directory:

aprogram.c	User program's source: opens, reads, writes to device
makefile	Command file that directs the **make** command
xyz.c	Source code for the device driver
xyz_cfg.c	Source code for the configure program

makefile for Sample XYZ Device Driver

This file, `makefile`, contains commands for building the sample XYZ device driver:

```
# Once this is done, have root user run xyz_cfg -q to query the kernel
# to see that the driver is not loaded.  Then run xyz_cfg -l to load it.
# Check again with xyz_cfg -q; if OK, then have non-root user run "aprogram."
# Then clean up by running xyz_cfg -u and verify absence of /dev/xyz.

#needed to get kernel services like devswadd()
KSYSLIST=/lib/kernex.exp

# needed for trace macro (containing a system call)
SYSLIST=/lib/syscalls.exp

all: aprogram xyz xyz_cfg

# import the kernel service calls and make xyzconfig() the entry point
xyz: xyz.o
     ld -e xyzconfig -o xyz -bI:$(KSYSLIST) -bI:$(SYSLIST) xyz.o

# It is necessary to use separate compile and link steps to avoid picking up
# routines like printf from libc.a. There is a kernel printf().
# _KERNEL needed to get trace macro, others determined by header files
xyz.o: xyz.c
     cc -c -D_ALL_SOURCE -D_POSIX_SOURCE -D_KERNEL xyz.c

# this is to create the driver's configure method that extends the kernel
xyz_cfg: xyz_cfg.c
     cc -o xyz_cfg xyz_cfg.c

aprogram: aprogram.c
     cc -o aprogram aprogram.c
```

Configuration Program for Sample XYZ Device Driver

This configuration program, `xyz_cfg.c`, is *not* recommended for configuring drivers in AIX. It avoids the use of ODM routines for simplicity.

```
/*
 * FUNCTION: Configure/Unconfigure program for bare bones driver, xyz
 *   Normally, one has a configure program (run at boot time)
 *   and then a separate unconfigure program (run interactively).
 *   Run "xyz_cfg" to see parameters needed.
*/

#include <stdio.h>           /* for printf() */
#include <unistd.h>          /* for getopt() */
#include <stdlib.h>          /* for exit() */
#include <sys/types.h>       /* for dev_t and other declarations */
#include <sys/errno.h>       /* for perror() */
#include <sys/sysmacros.h>   /* for makedev() */
#include <sys/sysconfig.h>   /* for sysconfig() */
#include <sys/device.h>      /* for CFG_INIT & other flags */
#include <sys/mode.h>        /* for mknod() */

void main(int argc, char *argv[])
{
  struct cfg_load cfg;          /* to load kernel extension */
  struct cfg_dd xyzcfg;          /* to invoke xyzconfig() */
  int majorno, minorno;
  dev_t device_number;
  int     ch;               /* flag char returned by getopt */

  extern  int     optind;         /* for getopt function */
  extern  char    *optarg;        /* for getopt function */

     /* normally call ODM routines to get values for these */
  majorno = 99;
  minorno = 0;
  device_number = makedev(majorno, minorno);

             /* parse command line */

  if(argc <= 1)
  { printf("You must give an argument.\n");
    printf("arguments to xyz_cfg are:\n");
    printf("\t-l to load the driver and invoke xyzconfig() \n");
    printf("\t-u to invoke xyzconfig() and unload the driver \n");
    printf("\t-q to query the status of the kernel extension\n");
  }

  while ((ch = getopt(argc,argv,"luq")) != EOF)
  { switch (ch)
    { case 'l':  /* load the driver--assume is first time (shouldn't!) */
     cfg.path = "./xyz";     /* path is local--usually /etc/drivers */
     if (sysconfig(SYS_KLOAD, &cfg, sizeof(cfg)) == -1)
     { perror("sysconfig SYS_KLOAD FAILED");
       exit(1);
     }
     xyzcfg.kmid = cfg.kmid; /* kernel module ID from SYS_KLOAD */
     xyzcfg.devno = device_number;
     xyzcfg.cmd = CFG_INIT;
     if (sysconfig(SYS_CFGDD, &xyzcfg, sizeof(xyzcfg)) == -1)
     { perror("sysconfig SYS_CFGDD FAILED");
       exit(1);
        }

     /* make /dev entry in honor of device switch table entry */
     if (mknod("/dev/xyz", 0666 | _S_IFCHR, device_number) == -1)
     { perror("mknod FAILED");
       exit(1);
        }

     break;
```

```
      case 'u':     /* unload the driver */

     /* the kmid lost once this exits, so we requery the information */
     cfg.path = "./xyz";
     if (sysconfig(SYS_QUERYLOAD, &cfg, sizeof(cfg)) == -1)
     { perror("sysconfig SYS_QUERYLOAD FAILED");
       exit(1);
     }

     xyzcfg.kmid = cfg.kmid; /* kernel module ID from SYS_QUERYLOAD */
     xyzcfg.devno = device_number;
     xyzcfg.cmd = CFG_TERM;
     if (sysconfig(SYS_CFGDD, &xyzcfg, sizeof(xyzcfg)) == -1)
     { perror("sysconfig SYS_CFGDD FAILED");
       exit(1);
     }

     /* remove /dev entry...normally would use ODM's reldevno() */
     unlink("/dev/xyz");

     if (sysconfig(SYS_KULOAD, &cfg, sizeof(cfg)) == -1)
     { perror("sysconfig SYS_KULOAD FAILED");
       exit(1);
     }
     break;

      case 'q':     /* query the status of the system call */
     cfg.path = "./xyz";
     if (sysconfig(SYS_QUERYLOAD, &cfg, sizeof(cfg)) == -1)
     { perror("sysconfig SYS_QUERYLOAD FAILED");
       exit(3);
     }
     printf("The kernel module ID is %d\n", cfg.kmid);

     xyzcfg.kmid = cfg.kmid; /* kernel module ID from SYS_QUERYLOAD */
     xyzcfg.devno = device_number;
     xyzcfg.cmd = CFG_QVPD;
     if (sysconfig(SYS_CFGDD, &xyzcfg, sizeof(xyzcfg)) == -1)
     { perror("sysconfig SYS_CFGDD FAILED");
       exit(1);
     }
     break;

      default:
        printf("arguments to xyz_cfg are:\n");
        printf("\t-l to load the driver and invoke xyzconfig() \n");
        printf("\t-u to invoke xyzconfig() and unload the driver \n");
        printf("\t-q to query the status of the kernel extension\n");
    } /* end switch on ch */
  } /* end while getopt */
  exit(0);
}
```

Source Code for Sample XYZ Device Driver

This file, `xyz.c`, contains source code for the sample device driver:

```
#include <sys/types.h>       /* for dev_t and other types */
#include <sys/errno.h>       /* for errno declarations */
#include <sys/sysconfig.h>   /* for sysconfig() */
#include <sys/device.h>      /* for devsw */
#include <sys/trchkid.h>     /* for trace hook macros */

/************************************************************
  BARE BONES DRIVER

This shows the format of a minimal set of entry points for a
pseudo-driver.
************************************************************/

/************** xyzopen ***********************/
int xyzopen(dev_t devno, ulong devflag, chan_t chan, int ext)
{
  TRCHKL5T(HKWD_USER1, 0x7, devno, devflag, chan, ext);
  return(0);
}

/************** xyzclose **********************/
int xyzclose(dev_t devno, chan_t chan)
{
  TRCHKL3T(HKWD_USER1, 0x8, devno, chan);
  return(0);
}

/************** xyzread ***********************/
int xyzread(dev_t devno, struct uio *uiop, chan_t chan, int ext)
{
  TRCHKL5T(HKWD_USER1, 0x9, devno, uiop, chan, ext);
  return(0);
}

/************** xyzwrite **********************/
int xyzwrite(dev_t devno, struct uio *uiop, chan_t chan, int ext)
{
  TRCHKL5T(HKWD_USER1, 0x0a, devno, uiop, chan, ext);
  return(0);
}
/************** xyzconfig **********************/
int xyzconfig(dev_t devno, int cmd, struct uio *uiop)
{ struct devsw dsw_struct;
  extern int nodev();
  int return_code;

  /* trace macro to print received parameters in hex */
  TRCHKL4T(HKWD_USER1, 0x1, devno, cmd, uiop);  /* 0x01-tracept label */

  switch(cmd)
  { case CFG_INIT:
      dsw_struct.d_open     = xyzopen;
      dsw_struct.d_close    = xyzclose;
      dsw_struct.d_read     = xyzread;
      dsw_struct.d_write    = xyzwrite;
      dsw_struct.d_ioctl    = nodev;
      dsw_struct.d_strategy = nodev;
      dsw_struct.d_ttys     = NULL;
      dsw_struct.d_select   = nodev;
      dsw_struct.d_config   = xyzconfig;
      dsw_struct.d_print    = nodev;
      dsw_struct.d_dump     = nodev;
      dsw_struct.d_mpx      = nodev;
      dsw_struct.d_revoke   = nodev;
      dsw_struct.d_dsdptr   = NULL;
      dsw_struct.d_opts     = NULL;
```

```
      if((return_code = devswadd(devno, &dsw_struct)) != 0)
      { TRCHKL3T(HKWD_USER1, 0x2, devno, dsw_struct);
     return(return_code);
      }

      /* entry points now in device switch table */
      break;


    case CFG_TERM:
      TRCHKL1T(HKWD_USER1, 0x3);
      if((return_code = devswdel(devno)) != 0)
      { TRCHKL2T(HKWD_USER1, 0x4, devno);
        return(return_code);
      }

      break;

    case CFG_QVPD:
      TRCHKL1T(HKWD_USER1, 0x5);/* would normally handle this case too */
      break;

    default:
      TRCHKL1T(HKWD_USER1, 0x6);
      return(EINVAL);
  } /* end switch(cmd) */
  return(0);
}
```

User Program to Invoke Sample XYZ Device Driver

This file, `aprogram.c`, contains sample user code to invoke the sample device driver:

```
#include <stdio.h>        /* for printf() */
#include <fcntl.h>        /* for open(), close() */
#include <unistd.h>       /* for read(), write() */
#include <sys/errno.h>    /* for perror() */

int fd;          /* file descriptor */
char buf[10];    /* read/write buffer */
void main()
{
  printf("buf pointer: 0x%x\n", buf);
  if((fd = open("/dev/xyz", O_RDWR)) == -1)
  { perror("open /dev/xyz FAILED");
    exit(1);
  }
  if(read(fd, buf, sizeof(buf)) == -1)
  { perror("read FAILED");
    exit(1);
  }
  if(write(fd, buf, sizeof(buf)) == -1)
  { perror("write FAILED");
    exit(1);
  }
  if(close(fd) == -1)
  { perror("close FAILED");
    exit(1);
  }
}
```

Running the Sample XYZ Device Driver

The lines prefixed by a # were commands executed as a user with root authority in a window. The lines prefixed by a > were commands executed as a staff user in another window. The order of the commands is as listed.

```
# trace -j'010' -l -s -a &
[1]     21971
# xyz_cfg -q
The kernel module ID is 0
sysconfig SYS_CFGDD FAILED: No such device
[1] + 21971      Done                    trace -j'010' -l -s -a &
# xyz_cfg -l
# xyz_cfg -q
The kernel module ID is 21790464
     > $ ls /dev/xyz
     > /dev/xyz
# chmod 666 /dev/xyz
     > $ aprogram
     > buf pointer: 0x200516c0
# trcstop
# trcrpt -O'exec=y' -O'pid=n' -O'svc=y' -O'timestamp=1' >
$HOME/frog
     > $ ls /dev/xyz
     > /dev/xyz
# xyz_cfg -u
# xyz_cfg -q
The kernel module ID is 0
sysconfig SYS_CFGDD FAILED: No such device
#
     > $ ls /dev/xyz
     > ls: 0653-341 The file /dev/xyz does not exist.
```

Trace Output for Sample XYZ Device Driver

Here is the output of **trcrpt** (abbreviated for space):

The hook data indicates calls to xyzconfig (load and query), open, read, write, and close.

```
ID  PROCESS NAME   I SYSTEM CALL     ELAPSED KERNEL  INTERRUPT

001 trace                           0.000000 TRACE ON channel 0
010 trace                          19.362228 UNDEFINED TRACE ID idx 0x21dc traceid
0010
        hookword 10E0000 type 0E
        hookdata 0000 00000001 00630000 00000001 2FF97F1C 00000000
010 trace                          25.716153 UNDEFINED TRACE ID idx 0x2230 traceid
0010
        hookword 10E0000 type 0E
        hookdata 0000 00000001 00630000 00000003 2FF97F1C 00000000
010 trace                          25.716159 UNDEFINED TRACE ID idx 0x224c traceid
0010
        hookword 10A0000 type 0A
        hookdata 0000 00000005
010 trace                          99.695506 UNDEFINED TRACE ID idx 0x24e8 traceid
0010
        hookword 10E0000 type 0E
        hookdata 0000 00000007 00630000 00000003 00000000 00000000
010 trace                          99.706120 UNDEFINED TRACE ID idx 0x2504 traceid
0010
        hookword 10E0000 type 0E
        hookdata 0000 00000009 00630000 2FF97DC0 00000000 00000000
010 trace                          99.706318 UNDEFINED TRACE ID idx 0x2520 traceid
0010
        hookword 10E0000 type 0E
        hookdata 0000 0000000A 00630000 2FF97DC0 00000000 00000000
010 trace                          99.706446 UNDEFINED TRACE ID idx 0x253c traceid
0010
        hookword 10E0000 type 0E
        hookdata 0000 00000008 00630000 00000000 00000000 00000000
002 trace                         109.684978 TRACE OFF channel 0
```

Routines on the Interrupt Side

When one of the system's processors receives an external interrupt, an AIX processor interrupt handler, for that particular interrupt level, begins execution. This portion of the AIX kernel determines which device interrupt handler, or which collection of handlers, to invoke. For more information on interrupt processing, see "Interrupts" on page 3-1.

A card on the system bus that serves as an adapter between the system and the device may generate an interrupt on the bus. Its device driver will need to configure a routine to handle that interrupt, and may wish to add other routines to handle *off-level* interrupts, which are scheduled by the device interrupt handler as a way to defer interrupt processing.

xyzintr Entry Point

This routine is registered to the kernel by a call to the **i_init** kernel service. The **xyzopen** routine typically calls **i_init**, and pins the bottom half of the device driver. Because routines executing on the interrupt side cannot afford to be preempted by demand paging, they are placed within the driver's bottom half (they are pinned in RAM).

This routine usually does the following:

- Determines the slot of the card that generated the device interrupt.

  This routine may also have to determine whether to process the interrupt. The interrupt level can be shared, so the interrupt may be intended for another device interrupt handler.

- Disables some other device interrupts to prevent this routine from being reentered if another interrupt for the same device driver is received.

  This technique of avoiding concurrent execution can fail on computers with multiple processors.

Because interrupt handlers have to address concurrency by serializing interrupt processing in some way, the handler's **latency (**the time between when control of the system processor is assumed and when control can be relinquished) must be minimized in order to maximize the system's ability to respond to other device interrupts.

- Transfers data from the device into a buffer, or notifies a user program that something has happened.
- Schedules an off-level interrupt handler to complete processing of input data.

  This can be done to minimize the handler's latency. As an alternative, the driver can depend on kernel processes to handle received data.

There are some special concerns that apply to routines on the interrupt side:

- Many kernel services can only be invoked from routines on the call side.
- These routines cannot go to sleep, though they can post events to (waken) routines on the call side.
- These routines cannot obtain or release locks.

  Routines on the call side that are in the bottom half of a driver can adjust locks, provided the events or lockwords are also pinned in RAM.
- An interrupt handler is called at the priority registered when the handler is configured into the kernel via the **i_init** kernel service. The handler's execution can only be preempted by interrupts with a higher priority.
- Interrupt handlers have volatile data (local variables) in a pinned stack that is less than 4K bytes in size.

  Note: Because **xmalloc** cannot execute on the interrupt side, any non-volatile buffers that an interrupt handler needs must be previously allocated on the call side.
- Routines that handle off-level interrupts run at a less favored interrupt priority.

xyzcallback Entry Point

This entry point is used only in device drivers for cards attached to a PCMCIA bus.

This routine is registered to the kernel by calling a PCMCIA Card Service. It executes in the context of a kernel process and does not need to be pinned in a usual environment. It does need to be pinned when a device driver gets a PM PAGE FREEZE NOTICE from power management threads until it gets a PM PAGE UNFREEZE NOTICE.

There is a special concern that applies to an **xyzcallback** routine:

- It cannot go to sleep for a long time, though it can post events to (waken) routines on the call side.

For more information, see "Configuration of Devices on PCMCIA Systems" on page 6-9.

Pinning Device Driver Object Files

Sometimes a driver routine (such as an interrupt handler) and any associated data is required to be kept in RAM. This requirement might exist to avoid handling a page fault so the routine can execute within a fixed period of time, or to avoid receiving a page fault while interrupts are disabled.

Typically, a device driver writer compiles or links all routine and data definitions into one loadable file (the bottom half of the device driver), because the **pincode** kernel service marks each page of the loaded object file as being required to be kept in RAM. The method for identifying the object file is that, a pointer to a routine within the object file is an input value for the **pincode** kernel service.

Routines and data that can be subject to page replacement are typically collected into another loadable object (the top half of the device driver).

Routines that wish to allocate buffers from the kernel heap (by calling the **xmalloc** kernel service) must take care which heap the allocation is from. Routines in the bottom half allocate data from the kernel's pinned heap; and routines in the top half allocate data from either heap. For more information on this, see "Memory Management" on page 4-1.

Any routine in the bottom half that invokes a routine in the top half, or refers to data in the top half, negates any purpose for having been pinned in RAM. The routine may (intermittently) cause a page fault.

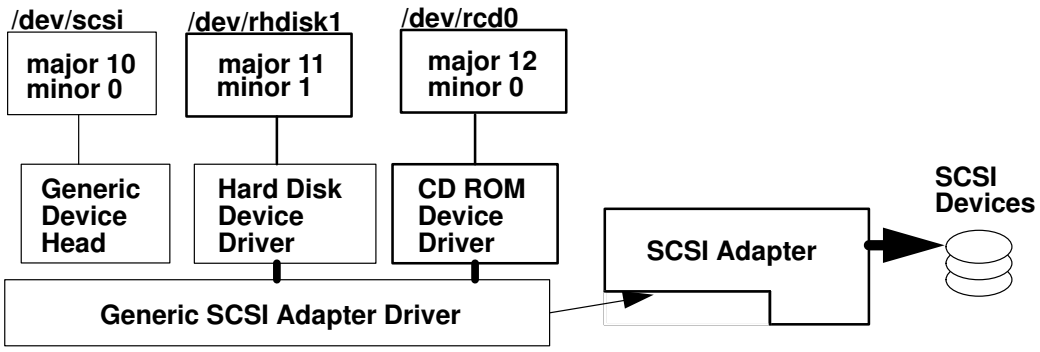
Typically, a device driver's open routine pins the device driver's bottom half when the device special file is first opened. The driver's close routine checks for any outstanding open calls on the file and then calls the **unpincode** kernel service.

Driving a SCSI Attached Device

A SCSI device driver converts I/O requests into SCSI commands, and passes these commands to the SCSI adapter driver provided in AIX. Therefore, a SCSI device driver only consists of routines that execute on the call side. The SCSI device driver passes commands to the adapter driver by direct invocation of its routines; therefore, a SCSI device driver must be integrated with the existing SCSI subsystem. The SCSI subsystem is illustrated in the SCSI Subsystem figure.

The generic SCSI device head associated with the special file `/dev/scsi#` (where `#` is 0, 1, 2, or some other number) permits a user program with root user authority to send SCSI commands through the associated SCSI adapter.

For more information on how to drive a SCSI attached device, see “SCSI Device Drivers” on page 8-1.



SCSI Subsystem

Other Topics

It may be necessary to integrate a device driver into some existing subsystem in AIX. For example, you may need to integrate a driver into one of the following subsystems:

- TTY subsystem

  See “STREAMS-Based TTY Subsystem Interface” on page 11-1 for more information on supporting terminal users through a serial device.

- Graphics device subsystem

  See “Implementing Graphical Input and 2D Graphics Device Drivers” on page 12-1 for more information on supporting a pointing device, graphics monitor, keyboard, or other graphics related device.

- TCP/IP

  See “Implementing a Network Device Driver” on page 13-1 for more information on supporting the TCP/IP socket interface or the Transport Layer Interface (TLI) through a network (Ethernet, Token Ring, or other) adapter.

- Socket Interface to your own protocol

  See “Network Interfaces and Protocols” on page 14-1 for more information on implementing a protocol that is an alternative to TCP/IP, which uses a network adapter driver provided by AIX.

Once a device driver is written and configured into the kernel, it is necessary to test and debug the driver routines. The example driver given in this chapter demonstrates the use of trace macros, but you may wish to incorporate error logging capability into a driver so that a system administrator can be notified of any irregularities.

Furthermore, defects in a driver could cause the system to halt after having copied an image of the kernel (called a *kernel dump*) to a logical volume reserved for that purpose. The kernel dump can be inspected with the **crash** utility. It may also be necessary to debug a driver using the kernel debugger, which comes with AIX and is configured into the kernel with the **bosboot** command. For more information on the use of such tools, see “Debugging Tools” on page 15-1.

Once a driver is ready for delivery, you may wish to archive the resulting binary and character files into a format that the **installp** command can process. For information on packaging files into **installp** format see “Software Product Packaging” in *AIX General Programming Concepts : Writing and Debugging Programs*.

In addition, you may wish to add an entry in the SMIT interface to enable users to install, configure, or remove your device in the same way that is done with devices supported by AIX. For more information on adding your own SMIT dialogs see “System Management Interface Tool (SMIT) for Programmers” in *AIX General Programming Concepts : Writing and Debugging Programs*.

Chapter 2. Device I/O

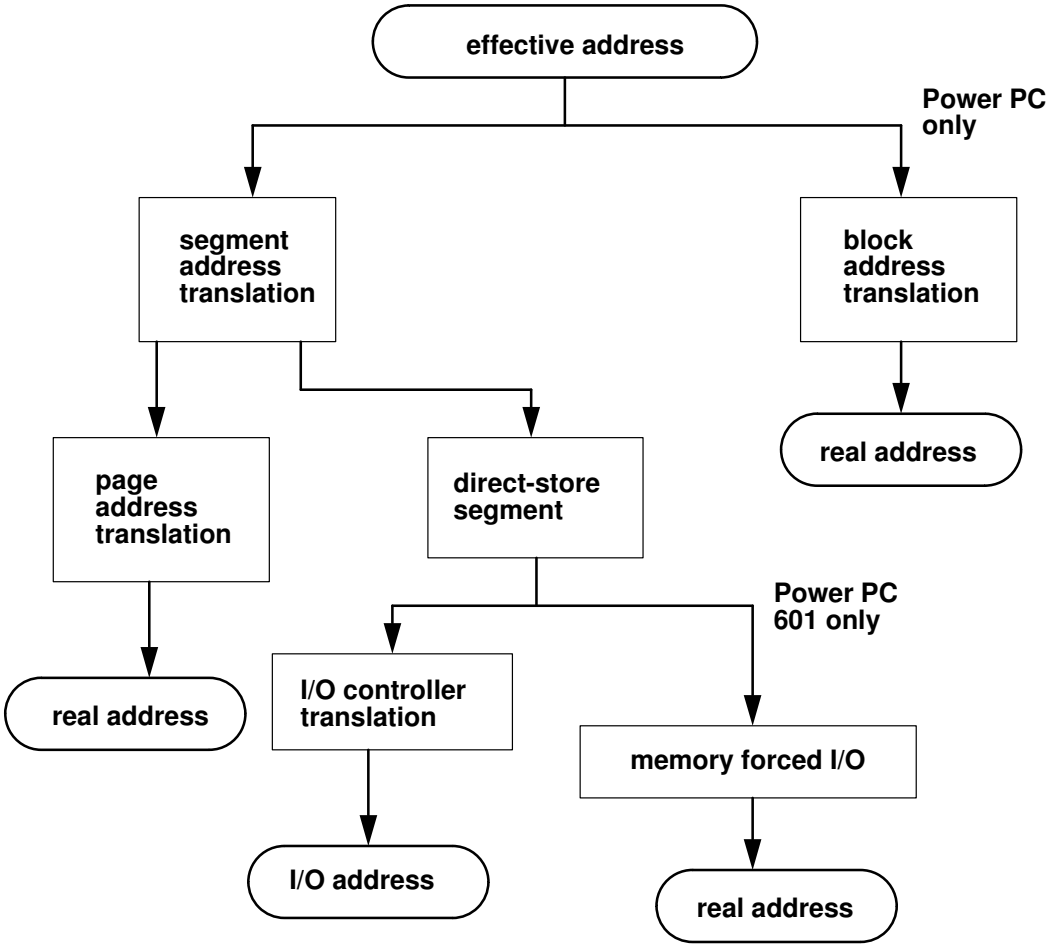
Even though a driver can perform many functions, one usually writes a driver to output data to a device or demand data from a device; in other words drivers usually perform device I/O. A driver may have to read from, or write to, registers on a card serving as an adapter between an I/O bus and a device connected to the card, or the driver may have to set up the means for data to be transferred in some other way.

Address Translation

There are two basic types of address translation implemented on system processors that AIX supports:

- Block address translation (on PowerPC only)
- Segment address translation

The various kinds of address translation are diagramed in the Address Translation figure.



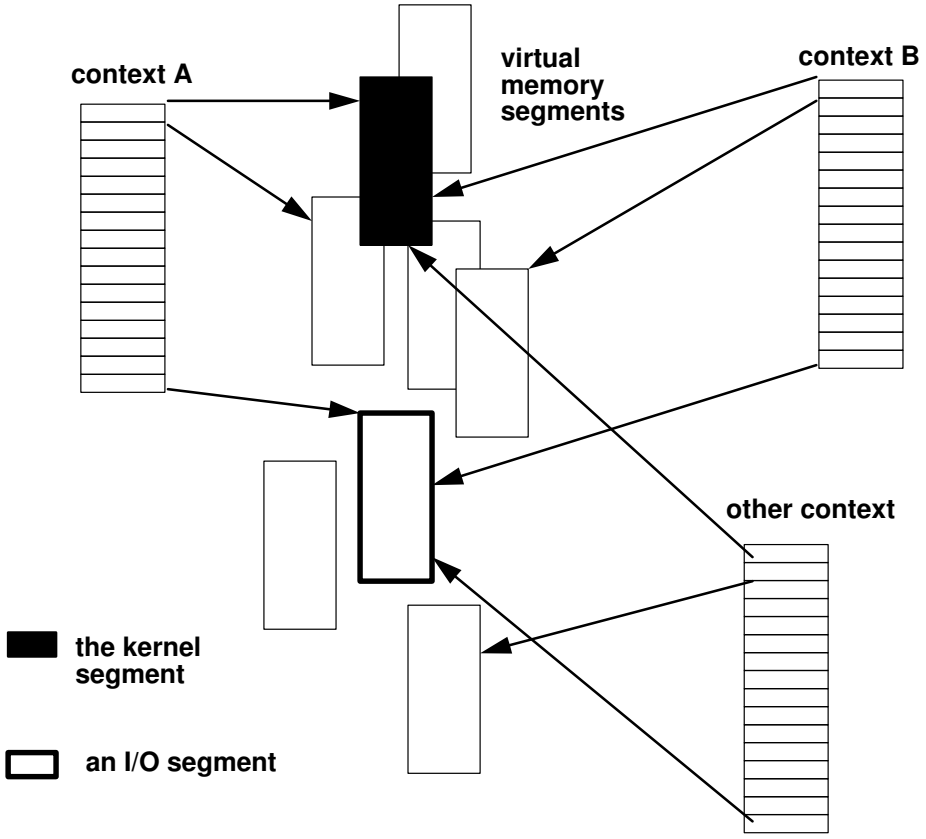
Address Translation

Block Address Translation

Block address translation (BAT) is a feature of the PowerPC architecture that is an alternative to page address translation. Pages have a fixed size, but blocks can be from 128 KB to 256 MB in size, although on the PowerPC 601 RISC Microprocessor, BAT areas can be no larger than 8MB. Whether this type of address translation is implemented in AIX depends on the system PowerPC processor. AIX does not use the BAT tables of the PowerPC 601 RISC Microprocessor, but instead uses memory-forced I/O, which is described later.

Segment Address Translation

All three processor architectures that AIX supports, POWER, POWER2, and PowerPC, can translate byte addresses by using sixteen segment registers. The contents of these registers, together with that of some other processor registers (such as the general purpose registers, instruction register, and machine state register) make up the context in which an instruction executes. Changing the contents of some of these registers (having saved the former values somewhere) is a *context switch*. A device driver's call side routines typically execute in the context of a process, and its interrupt side routines execute in the context of an interrupt handler. The Segmented Virtual Memory figure shows how virtual memory consists of segments accessed by instructions executing in various contexts.



Segmented Virtual Memory

In AIX, instructions typically execute in a context whose segment registers contain the following:

Segment register 0	Kernel segment identifier (ID) (Shared by all)
Segment register 1	Text segment ID (Where instructions are fetched from)
Segment register 2	Data segment ID (Not shared by others)
Segment register 13	Shared text segment ID (For shared libraries)
Segment register 15	I/O segment ID (For I/O. Often used, but not required.)

For more details, see “Program Address Space Overview” in the chapter about shared libraries and shared memory in *AIX General Programming Concepts : Writing and Debugging Programs*.

A segment register contains a segment identifier (segment ID) which determines, among other things, what kind of address translation is to be performed on addresses within that segment.

Consider the following kinds of segment address translation:

- Page address translation
- I/O controller interface translation
- Memory forced I/O (PowerPC 601 RISC Microprocessor)

Page address translation is usually performed on an address referenced by an instruction accessing system RAM. For the POWER and POWER2 architectures, this kind of address translation is detailed under “Memory Addressing” in the chapter about system processors in *Hardware Technical Information-General Architectures*. For the PowerPC architecture, page address translation is detailed in *PowerPC Architecture*.

A segment address is page translated if its corresponding segment register has the highest order bit clear (zero); otherwise, the address is given to the I/O controller for translation. The PowerPC 601 RISC Microprocessor has a feature, called memory-forced I/O, that enables the I/O controller to generate a real address. This address can be used when I/O space is memory mapped.

I/O Controller Types

Each processor architecture expects an I/O controller to interface between the system bus (the one the processors use to access RAM) and an I/O bus. A computer system may have more than one I/O bus, but each bus has its own I/O controller.

The following table shows processor types, I/O controllers, and I/O bus protocol combinations that are supported by AIX Version 4.1:

Processor	Controller	Bus Protocol	Address Space for I/O
POWER RS1	IOCC	Micro Channel	I/O address
POWER RSC	IOCC	Micro Channel	I/O address
POWER RS2	XIO	Micro Channel	I/O address
PowerPC	ASIC	Micro Channel	I/O address
PowerPC	PCIB/MC	PCI	real address (memory mapped I/O)

The following list explains some terms used in the preceding table:

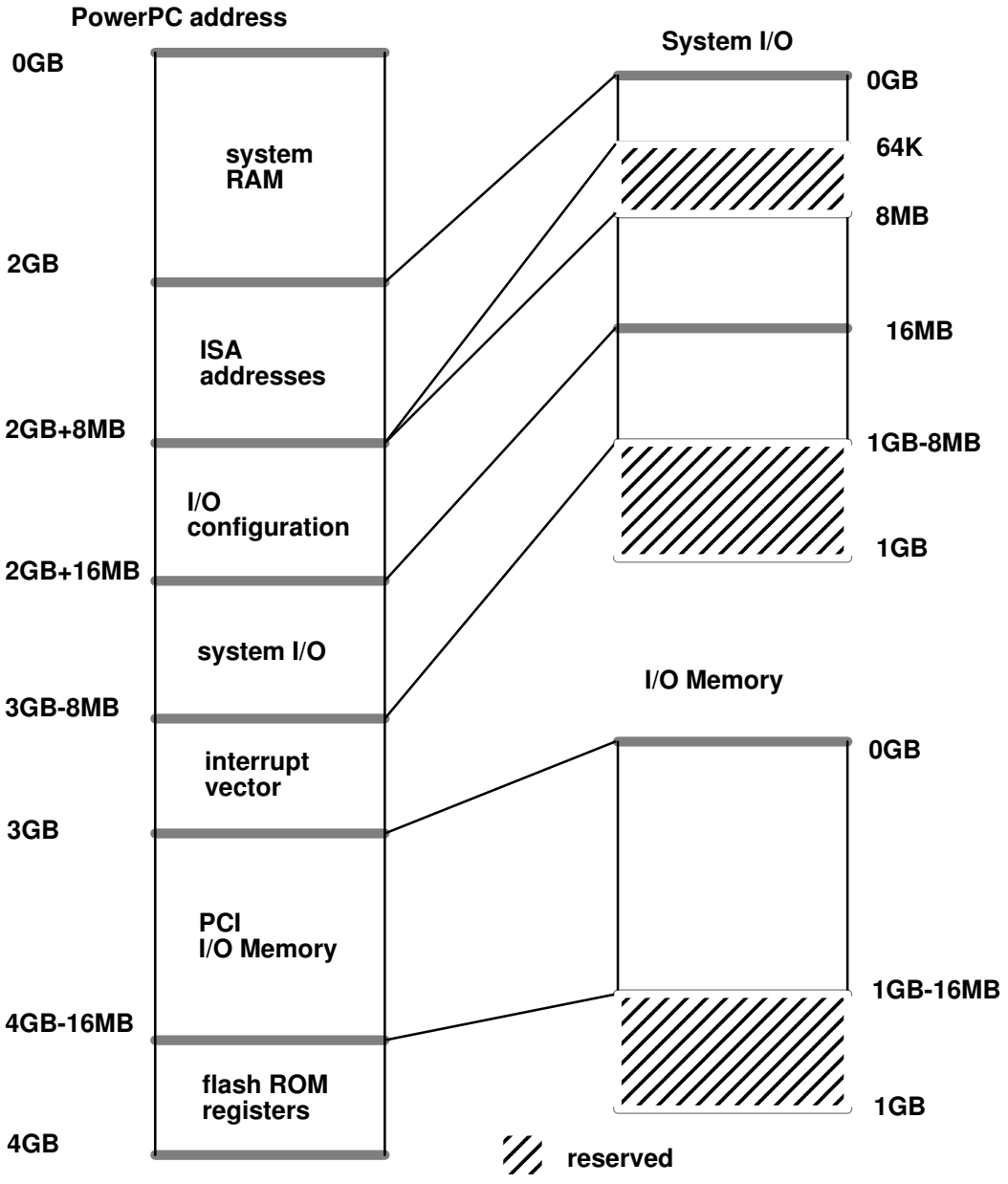
- Power RSC is POWER architecture on a single chip.
- Power RS2 is POWER2 architected chip.
- IOCC is I/O Channel Controller chip.
- XIO is Extended IOCC.
- ASIC is Application Specific Integrated Circuit chip.
- PCIB/MC is Peripheral Component Interconnect (PCI) Bridge/Memory Controller complex.

The first four configurations, which interface to a Micro Channel bus, are quite similar. I/O space is accessed by passing an address to an I/O controller for translation; the address is then an *I/O address*. In this case, I/O space is mapped to I/O addresses by the I/O controller.

Configurations having a PCIB/MC, which interfaces to a Peripheral Component Interconnect (PCI) bus, are quite different. The I/O space is part of real address space, meaning that a memory controller translates real addresses so that certain address ranges access system RAM, and other address ranges access other system or bus attached devices. In this sense, I/O space is memory mapped. Different systems may have different address ranges mapped to different devices. For implementation details, see the hardware technical reference for the particular system.

I/O Space on PCI and ISA Systems

On PowerPC systems that do not have a Micro Channel bus, I/O space is memory mapped by the memory controller. So, I/O space is accessed by generating real addresses in one of two ways: either through block address translation, or memory-forced I/O (PowerPC 601 RISC Microprocessor only). The Controller's Memory Map (32 bit) figure shows an example of how the memory controller maps real addresses to bus addresses.



Controller's Memory Map (32 bit)

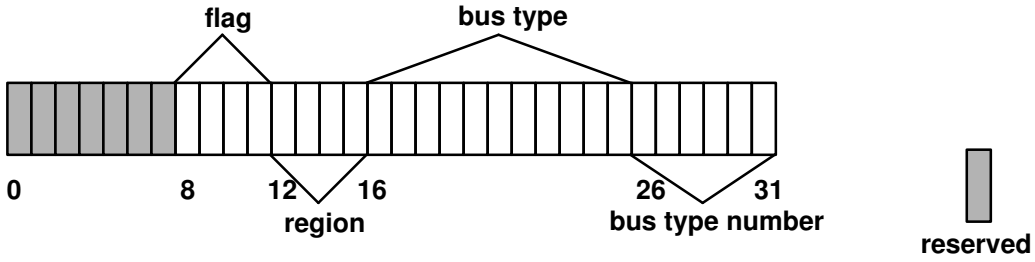
Programmed I/O to PCI, ISA, and PCMCIA Devices

A routine is said to perform *programmed I/O* whenever it issues a load or store instruction with an address mapped to a bus or planar device. One distinguishes programmed I/O, where a system processor performs the data transfer, from direct memory access (DMA) where data is transferred by some other means.

For example, to output the value `0x12345678` to some register at offset `0x2f7`:

```
volatile uchar *ioaddr;
struct io_map io_map;
...
ioaddr = iomem_att(&io_map);    /* do after io_map initialized */
*(ioaddr + 0x2f7) = 0x12345678;
eieio();                /* ensures I/O instructions complete */
...
iomem_det(ioaddr);
```

The argument to **iomem_att** is a pointer to a structure **io_map** as defined in **sys/ioacc.h**. The calling routine provides the size of the address space needed, and a bus ID which specifies the bus type of region to be mapped. The figure Format of a Bus ID (real address, 32 bit) shows the format of a bus ID. The device driver's configuration entry point must get a valid bus ID from its configure method.



Format of a Bus ID (real address, 32 bit)

In this case, the bus type might be IO_ISA. The region might be ISA_IOMEM if I/O is to an ISA adapter's registers, or ISA_BUSMEM if I/O is to RAM on an ISA adapter.

A call to **iomem_att** returns a bus address in **io_map.** The bus address is to be passed to **iomem_det** after the I/O is complete.

There is no exception handling for programmed I/O on systems that do not have Micro Channel. An I/O exception causes the system to halt.

For PCMCIA devices, the bus type can be IO_ISA or IO_PCI. The bus type depends on the bus that the PCMCIA bus is attached to. Before **iomem_att** is called, a call to RequestIO for I/O memory or RequestWindow for bus memory is required.

Direct Memory Access

A method of transferring data to or from a device without having a system processor issue load or store instructions is to use *direct memory access* (DMA), which relies on capabilities designed into the DMA controller and the adapter interfacing to the attached device.

There are two types of DMA:

- DMA Master

  When an adapter arbitrates for the bus, and is able to transfer data directly by generating its own bus addresses and transfer lengths, then the transfer is *DMA master*, and the card is a *DMA master adapter*.

- DMA Slave

  When an adapter arbitrates for control for the bus, but lacks the ability to generate its own bus addresses to perform data transfer, so that a third party (a DMA controller) performs the data transfer, then the transfer is *DMA slave*, and the card is a *DMA slave adapter*.

  Because bus addresses are meaningless on this system during DMA slave operations, DMA slave adapters cannot use the addresses during the DMA transfer to indicate the intended location for the data. Because of the programmed I/O commands that the device driver has previously issued to the adapter, DMA write operations indicate where the adapter should put the data. The DMA read operations indicate where the adapter should retrieve the data. The adapter typically knows where the data is to be put (during a DMA write operation) or where it is to come from (during DMA read operations).

The steps for performing DMA differ between bus types, though the steps for performing DMA on ISA are similar to those for PCI. There is currently no DMA support for PCMCIA adapters.

DMA on POWER and POWER2 Architectures

The RIOS-1 and RIOS-2 architectures are not cache-consistent. Therefore, on these platforms, memory pages involved in a DMA transfer have to be made inaccessible to the processor (hidden), and the processor data cache must be flushed accordingly.

The IOCCs on these models are buffered (including the dual-buffered XIO) implementations, and thus require the use of kernel services to flush and invalidate buffers.

DMA on RSC (Single-Chip) Architectures

The RSC architecture is a cache-consistent architecture; therefore page hiding and data cache flushes aren't necessary.

The IOCC on this architecture is non-buffered, therefore does not require buffer flushes or buffer invalidation.

DMA on PowerPC Architectures

The PowerPC architecture is a cache-consistent architecture, therefore page hiding and data cache flushes aren't necessary. However, if a DMA operation modifies an instruction stream, instruction cache management is required.

DMA Routines for PCI and ISA Devices

The **dio** structure, which is defined in **sys**/**dma.h**, has many uses by a device driver. It is used both to pass a list of virtual addresses and lengths of buffers to the **d_map_list** and **d_map_slave** services, and to receive the resulting list of bus addresses (**d_map_list** only) for use by the device in the data transfer. Note that for calls to **d_map_slave**, the driver does not need a **dio** bus list, since by nature the address generation for slaves is hidden.

Typically, a device driver will provide a **dio** structure containing only one buffer and length in the list. Then, if the buffer length spans many pages, the bus address list would contain multiple entries reflecting the physical locations making up the virtually contiguous buffer. It may be desirable to process several buffer requests in one DMA operation. A driver can group multiple requests by specifying a virtual list in a **dio** structure.

The device driver must allocate any pinned storage for all **dio** lists needed. The driver will need at least two **dio** structures, one for passing in the virtual list, and another for accepting the resulting bus list. The driver can have many **dio** lists if it plans to have multiple outstanding I/O commands to its device. The length of each list is dependent on the function of the device and driver. The virtual list needs as many elements as are planned to be coalesced into one operation by the device. A formula for estimating how many elements the bus address list will need is the sum of each of the buffer lengths divided by page size plus 2. One way to represent this formula is:

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

This is to handle a worst case situation, where, for a contiguous buffer spanning multiple pages, each physical page is discontiguous, and neither the starting or ending addresses are page-aligned.

If the **d_map_list** service runs out of space while filling in the **dio** bus list, then an error, DMA_DIOFULL, is returned to the device driver and the `bytes_done` field of the **dio** virtual list is set to how many bytes were successfully mapped in the bus list. This byte count is a multiple of the `minxfer` field provided to the **d_map_list** or **d_map_slave** service. Also, the `resid_iov` field of the virtual list is set to the index of the first **d_iovec** entry representing the remainder of **iovec**s that could not be mapped. The device driver then can initiate a partial transfer on its device and leave the remainder on its device queue, or make another call to the **d_map_list** with new **dio** lists for the remainder, then setup its device for the full transfer that was originally intended. If the driver chooses not to initiate the partial transfer, it still must make a call to **d_unmap_list** to undo the partial mapping.

If **d_map_list** or **d_map_slave** encounter an access violation on a page within the virtual list, then an error, DMA_NOACC, is returned to the device driver, and the `bytes_done` field of the **dio** virtual list is set to the number of bytes that preceded the faulting **iovec**. In this case, the `resid_iov` field is set to the index of the **d_iovec** entry that encountered the violation. From this information, the driver can determine which buffer contained the faulting page and report the failure of the request associated with the buffer. Note that in the DMA_NOACC case, the **bytes_done** count is *not* always a multiple of the `minxfer` field provided to the **d_map_list** or **d_map_slave** service, and no partial mapping is done. For slaves this means that no setup of the address generation hardware has been done. For masters this means the bus list is undefined. If the driver desires a partial transfer, it must make another call to the mapping service with the **dio** list adjusted to not include the faulting buffer.

Finally, if while mapping a transfer, either the **d_map_list** or **d_map_slave** services run out of resources to map the transfer, an error, DMA_NORES, is returned to the device driver. In this case, the `bytes_done` field of the **dio** virtual list is set to the number of bytes that were successfully mapped in the bus list. This byte count is a multiple of the `minxfer` field provided to the **d_map_list** or **d_map_slave** service. Also, the `resid_iov` field of the virtual list is set to the index of the first **d_iovec** of the remaining iovecs that could not be mapped. The device driver then can initiate a partial transfer on its device and leave the remainder on its device queue, or choose to leave the entire request on its device queue

and wait for resources to free up (for example, after device interrupt from previous operation). If the driver chooses not to initiate the partial transfer, it still must make a call to **d_unmap_list** or **d_unmap_slave** (for slaves) to undo the partial mapping.

The maximum value for the **minxfer** parameter of the **d_map_list** and **d_map_slave** services defaults to 64K bytes. This should be sufficient for almost all devices, but some may require a larger absolute minimum transfer. The maximum **minxfer** value can be increased by using the **DMA_MAXMIN_*** set of flags defined in **sys**/**dma.h**. One of these flags may be ORed into the `flags` field of the **d_map_init** service to increase the maximum possible **minxfer** value for subsequent calls to **d_map_list** and **d_map_slave**.

The only field of the bus list that a device driver modifies is the `total_iovecs` field to indicate how many elements are available in the list. The device driver never modifies any of the other fields in the bus list. It is this list that the device driver uses to setup its device for the transfer, and it is this list that is provided to the **d_unmap_list** service to unmap the transfer. The **d_map_list** service sets the `used_iovecs` field to indicate how many elements it filled out.

As far as the virtual list, the device driver sets up all of the fields except for the `bytes_done` and `resid_iov` fields which are set by the mapping service.

To enhance portability, these services are typically invoked by using macros (such as D_MAP_INIT and D_MAP_LIST) defined in /**sys**/**dma.h**.

Page Protection

Page protection checking is performed by the **d_map_page**, **d_map_list**, and **d_map_slave** services by calling the **xmemdma** kernel service for each page of a requested transfer. If the intended direction of a transfer is device to memory, then the page access permissions must allow writing to the page. If the intended direction of a transfer is from memory to device, then the page access permissions need only allow reading from the page. If there is a protection violation, no mapping for the DMA transfer is performed, and an appropriate error code is returned.

The DMA_BYPASS flag allows a device driver to bypass the access checking functionality of these services; this flag should only be used for global system buffers such as mbufs or other command, control, and status buffers used by a device driver.

Peer-To-Peer DMA Support

Peer-to-Peer DMA is DMA transfer from one adapter's bus memory to another's, controlled by either a Bus Master or a DMA Slave with assistance of the I/O Controller. In either case, the transfer is independent of the CPU and does not involve system memory.

The flag BUS_DMA supports peer-to-peer DMA operations on calls to **d_map_page**, **d_map_list** and **d_map_slave**. This flag ensures that these services do not translate the provided address as a virtual address; instead it ensures that they treat the address as a bus address.

DMA Master I/O for an ISA Adapter

Because an ISA adapter can only generate 24 bit addresses, it can address up to 16MB. A device driver may need to ensure that DMA buffers are in low memory, unless there is no more than 16MB of RAM on the system. A driver allocates such buffers with the **rmalloc** kernel service, and later frees them with the **rmfree** kernel service. Then, the driver can "bounce" data from user supplied buffers (which may not be in low memory) into the buffer allocated by **rmalloc**. Then the driver could cause the data to be transferred to the ISA adapter with DMA.

For example, if the driver is to transfer data from a user buffer to a DMA master adapter, some steps for the transfer could be:

Steps for ISA DMA Master Transfer	
Module	**What Device Driver Could Do**
xyzopen	**rmalloc** driver buffer(s). Call DIO_INIT to allocate and initialize a **dio** structure. Call D_MAP_INIT which returns "d_handle" containing bus specific services. Call D_MAP_ENABLE if DMA not already enabled for this "d_handle". Call D_MAP_PAGE if **rmalloc**'ed buffer contained within a 4K page, or call D_MAP_LIST if **rmalloc**'ed buffer(s) span multiple pages.
xyzread or xyzwrite	Perform PIO write to adapter register to start DMA transfer. Copy data to bounce buffer (writes only).
xyzintr	(Interrupt received from adapter when DMA transfer complete.) Copy data out of bounce buffer (reads only). Perform PIO read of adapter register to check DMA status.
xyzclose	Call D_UNMAP_PAGE or D_UNMAP_LIST. Call D_MAP_DISABLE if DMA not already disabled for this "d_handle". Call D_MAP_CLEAR to free "d_handle" structure. Call DIO_FREE to free the **dio** structure, then **rmfree** the driver buffer(s).

Here is an example in pseudo-code of how a DMA master transfer could be done:

```
Initialization entry point: (xyzopen or xyzconfig)

 determine bus type for device from configuration information
 determine 64 vs.32 bit capabilities from configuration
           information
 call "handle = D_MAP_INIT(bid, DMA_MASTER|flags,
                           bus_flags, channel)"
 if handle == DMA_FAIL
        could not configure
 else
        call "D_MAP_ENABLE(handle)" (if necessary)

start_io entry point: (xyzread or xyzwrite)

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

finish_io entry point: (xyzintr)

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

unconfigure code: (xyzclose or xyzconfig)

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

DMA Slave Transfers on an ISA Adapter

Here is an example in pseudo-code of how a DMA slave transfer could be done:

```
initialization entry point: (xyzopen or xyzconfig)

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

start_io entry point: (xyzread or xyzwrite)

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

finish_io entry point: (xyzintr or xyzclose)

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

unconfigure entry point: (xyzclose or xyzconfig)

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

DMA Master Transfers on a PCI Adapter

The main difference between DMA on ISA and DMA on PCI is that there is no need for “bounce” buffers for PCI DMA; also, there are no DMA slave adapters on a PCI bus.

For example, if the driver is to transfer data from a user buffer to a DMA master adapter, some steps for the transfer could be:

Steps for PCI DMA Master Transfer (Short Term)	
Module	**What Device Driver Could Do**
xyzopen	Call DIO_INIT to allocate and initialize a **dio** structure. Call D_MAP_INIT which returns “d_handle” containing bus specific services. Call D_MAP_ENABLE if DMA not already enabled for this “d_handle”.
xyzread or xyzwrite	Call D_MAP_PAGE if buffer contained within a 4K page, or call D_MAP_LIST if buffer(s) span multiple pages. Perform PIO write to adapter register to start DMA transfer.
xyzintr	(Interrupt received from adapter when DMA transfer complete.) Perform PIO read of adapter register to check DMA status. Call D_UNMAP_PAGE or D_UNMAP_LIST.
xyzclose	Call D_MAP_DISABLE if DMA not already disabled for this “d_handle”. Call D_MAP_CLEAR to free “d_handle” structure. Call DIO_FREE to free the **dio** structure.

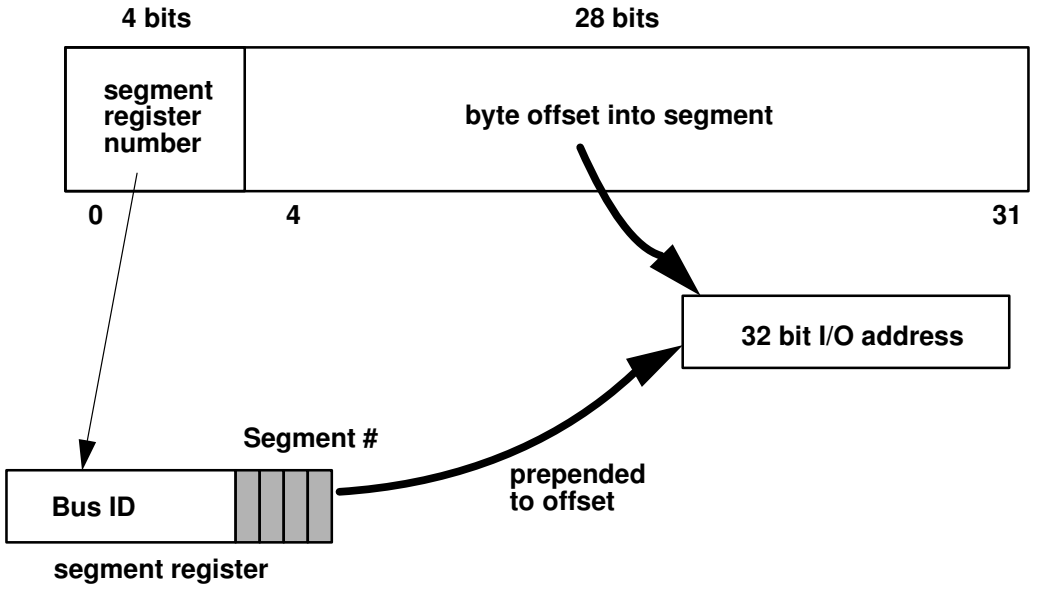
I/O Controller Interface Translation on Micro Channel Systems

A processor associates device registers or memory to particular addresses via *I/O controller interface translation*. When a processor instruction accesses an I/O bus, the address referenced by that instruction is said to *access an I/O controller interface* because all three processor architectures expect an I/O controller to interface between a processor and an I/O bus associated with that I/O controller.

I/O controller interface translation, like page address translation, is affected by the value of the segment ID contained in a segment register.

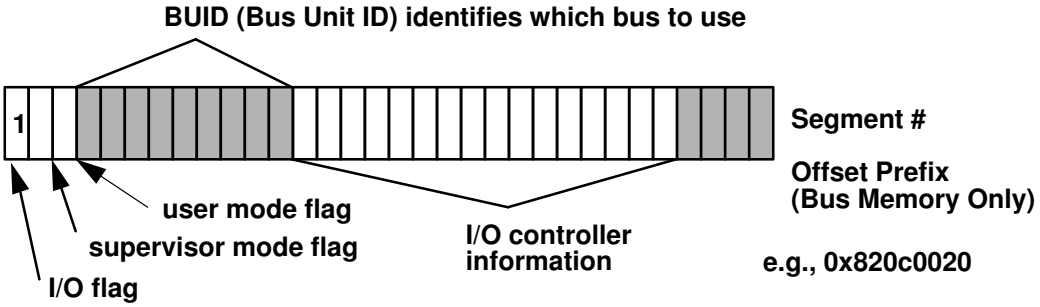
One performs I/O by executing load or store instructions referencing virtual addresses. Some other architectures (for example, the Intel x86 series) rely on special instructions to perform I/O to particular ports. The three processor architectures considered here translate addresses so that system device registers, adapter registers, adapter RAM, and so on, appear to be part of virtual memory.

The figure How an Address Specifies a Segment Register shows how the address specifies which segment register to use. For example, the address 0x50285740 says to look in segment register 5 for the segment ID, and it references byte 0x285740 within that segment. If the high order (leftmost) bit in segment register 5 is 0, then the address is translated as a page address, otherwise, the address is translated so that one can access an I/O bus through its I/O controller interface.



How an Address Specifies a Segment Register

A bus ID is a segment identifier associated with an I/O segment. The Format of a Bus ID figure shows the overall format of a bus ID contained in a segment register. Note that the high order bit is set to 1. This bit distinguishes which kind of address translation to perform: when set, it signifies that addresses in this segment require I/O controller interface translation.



Format of a Bus ID

I/O controller interface translation differs from page address translation in that no page tables are searched. So there is no way to protect one program accessing an I/O bus from another program accessing the same bus.

The bus identifier (*bid*) is a long value that is supplied to the **BUSIO_ATT**, **BUSMEM_ATT** and **IOCC_ATT** macros to obtain access to the system I/O bus resources and the Micro Channel bus. The bus identifier is also required when directly using the underlying **io_att** kernel service to obtain access to the bus, instead of using the macros. However, use of the macros is strongly advised. This bus identifier must be constructed by the device driver that is to gain access to the bus.

The bus identifier consists of:

- Bus unit number that is typically supplied in the device-dependent structure (DDS) by the device Configuration method
- Bus resource that is to be addressed
- Bus addressing modes to be used

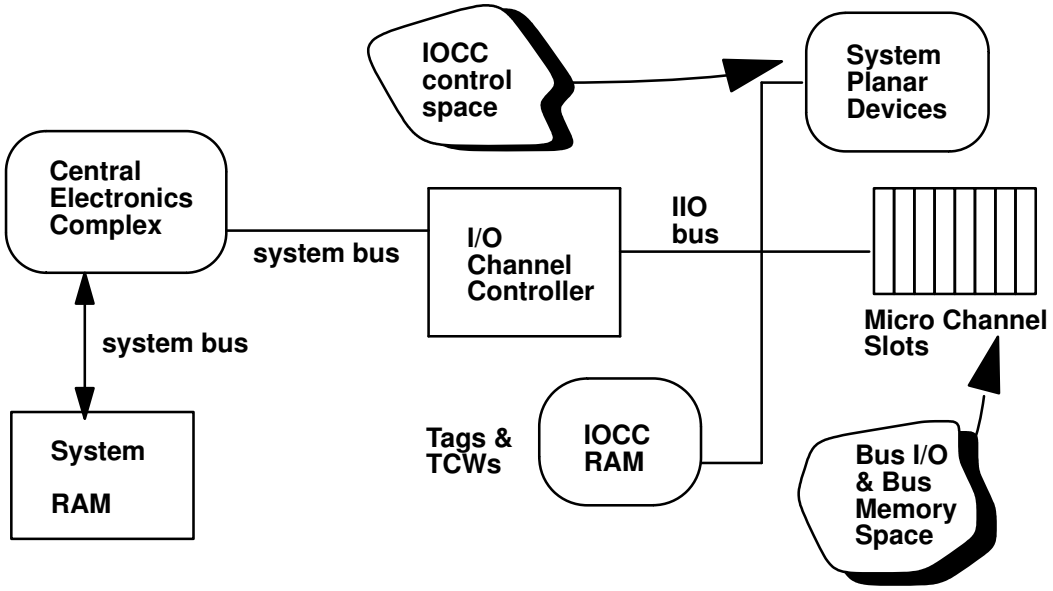
The device driver should construct the bus identifier by starting with the bus unit value provided in the DDS and then ORing in the additional bits required for proper bus access. For example, the bus unit number for bus 0 on the system is hex 82000000.

Note: This number should not be hardcoded but should be obtained by the adapter's Configuration method from the parent's **bus_id** attribute in the ODM database. The device driver should then OR in the following bits: hex 000C0020. This sets up a typical access mode with address checking enabled, address increment enabled, and TCW bypass enabled.

Warning: Device drivers that rely on the ability to disable either or both of the Address Check and Address Increment bits will not work on PowerPC machines. The PowerPC I/O architecture does not support the Address Check, Address Increment, and TCW Bypass bits. By default, these functions operate as enabled, and it is not possible to disable them.

I/O Address Spaces on Micro Channel Systems

The I/O Subsystem figure shows how portions of the I/O subsystem work with one another. Note that a system processor does not access an I/O bus directly, but relies on a special-purpose processor, mounted on the system planar with its own dedicated RAM, to serve as an interface between the system bus, which accesses system RAM, and the I/O bus.



The I/O Subsystem

Note that there are devices on the system planar itself, and there are devices plugged into slots accessing the Micro Channel bus. There are also resources on the IOCC itself that one may wish to modify or inspect. There is a distinction between I/O to a system planar device (or the IOCC itself) and I/O to a device attached to the Micro Channel bus. You specify which kind of I/O to do by selecting an I/O address segment, or “I/O space.”

There are several kinds of I/O address spaces defined by each processor architecture. The ones that affect a device driver are:

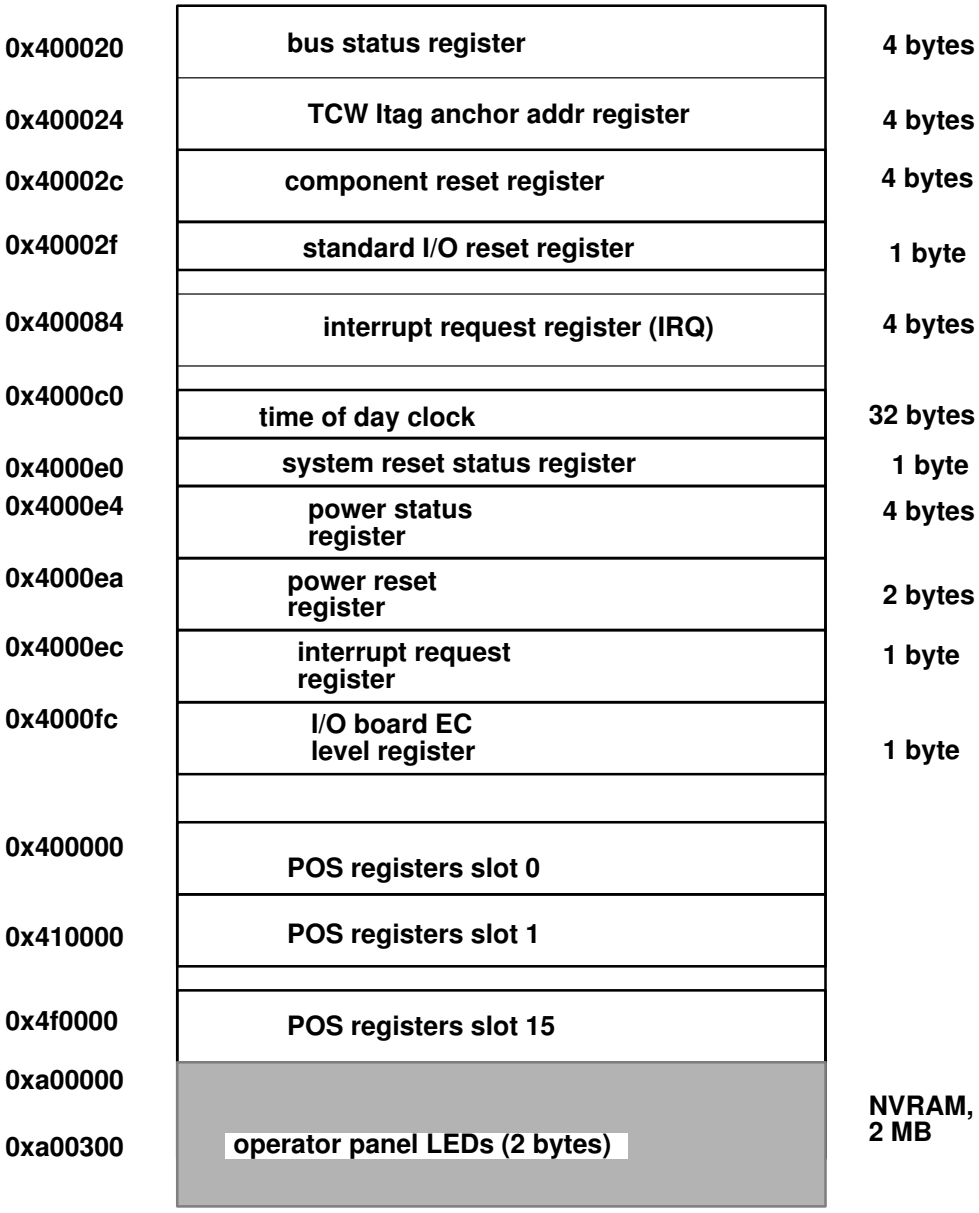
- IOCC control space (or system I/O space)
- Bus I/O space
- Bus memory space

Devices that are attached to the system planar board (also known as the *motherboard*) have addresses mapped to IOCC control space. Examples of such devices are: system timer, calendar, non-volatile RAM, the LED registers, programmable option select (POS) registers, and other planar device registers.

IOCC control space consists of one I/O segment called the IOCC control segment. An I/O address translates to this segment if the IOCC bit in the IOCC controller information part of a bus ID is set to 1. See the IOCC Control Space (Portion) figure for an example (for the POWER architecture) of what system planar devices are mapped to which addresses.

Both bus I/O space and bus memory space are collectively referred to as *standard I/O space*, and are accessed within one segment.

Any attempt to access IOCC control space without system privilege (as marked in the bus ID) causes an I/O exception.

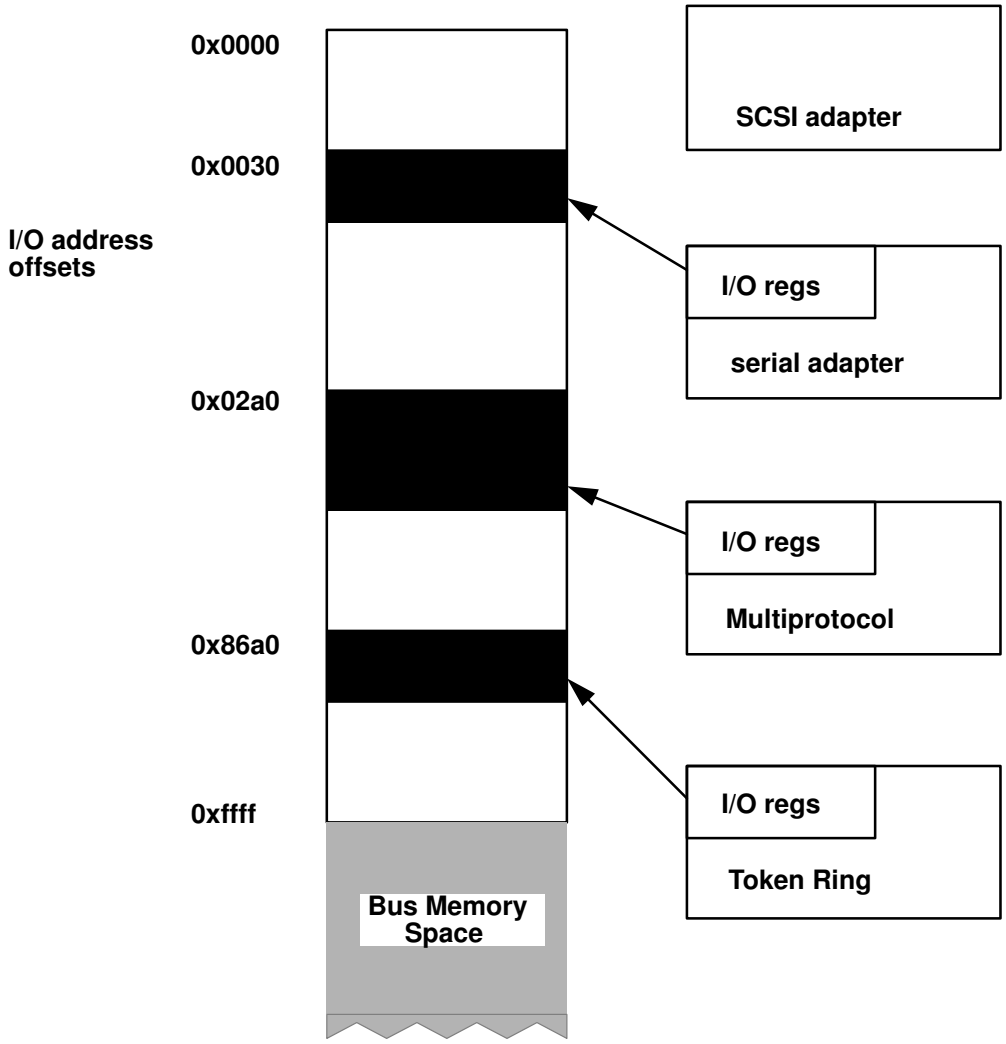


IOCC Control Space (Portion)

Note: Some of the information in the preceding IOCC Control Space (Portion) figure applies only to the POWER architecture and not to the PowerPC architecture.

So, for example, the address 0xf0400020 refers to BUID in register 15, where the offset specifies the bus status register. Note that IOCC RAM is mapped to IOCC control space.

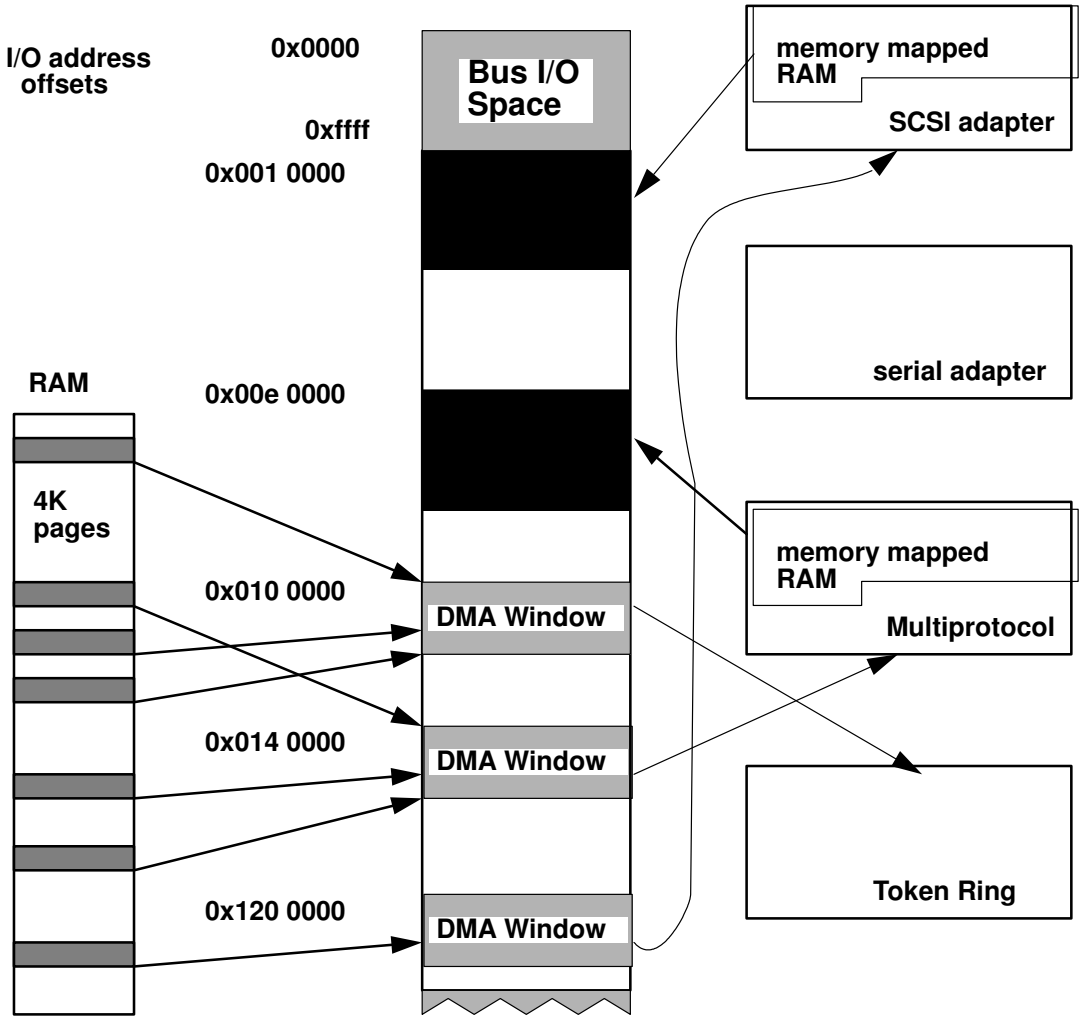
The register addresses start from the value of IO_IOCC as found in the header file /**usr/include/sys/iocc.h**. Here the value of IO_IOCC is 0x00400000 (4 MB).



Bus I/O Space (Example)

The Bus I/O Space (Example) figure shows an example of how adapters attached to the I/O bus can be mapped to bus I/O space. The IOCC checks each access to bus I/O space to see if the address is within range of some region mapped in bus I/O space. If not, it generates an I/O exception.

Note that the SCSI adapter has no registers mapped to addresses in bus I/O space, but the others do. For example, a load instruction from address 0x50000030 would (if segment register 5 has a bus ID for the standard I/O segment) get four bytes from the serial adapter's first register. Provided that segment register 15 has a standard I/O segment's bus ID, a store instruction to address 0xf00086a0 would place four bytes of data in the first register on the Token-Ring adapter.



Bus Memory Space (Example)

Each 4K page of bus memory space that is mapped for translation is associated with a Translation Control Word (TCW), which is a data structure kept in RAM dedicated for use by an IOCC. Any attempt to read or write to an address that is not within a page associated with a TCW causes an I/O exception. For more information on TCWs and protection, see "Translation, Protection, and the TCW Table," in "Programming Model: System I/O Structure" in *Hardware Technical Information-General Architectures*.

The Bus Memory Space (Example) figure shows an example of how the same four adapters have bus memory addresses mapped to RAM either on the adapters or on the system itself. Note that in this case, the serial adapter has no addresses in bus memory space for its use.

In this example, a load instruction from address 0xf0010000 would read the first four bytes in RAM on the SCSI adapter. Similarly, a store instruction to address 0xf00e0000 would write four bytes to RAM on the multiprotocol adapter.

Once the mapping is in place, transfer from system memory to adapter RAM amounts to issuing a sequence of load and store operations from one region of bus memory space to another.

Programmed I/O to Micro Channel Adapters

Programmed I/O requires a properly formatted bus identifier in a segment register. The device driver's configuration method gets a bus identifier for its particular bus by reading its parent `bus_id` attribute out of the predefined attribute (PdAt) object class of the ODM. The configuration method passes the bus identifier, within the device dependent structure (DDS), to the device driver's configuration entry point where the bus identifier is further modified as needed. For example, a driver may wish to set the bus ID bits associated with *privilege key* and *IOCC space select* by the following statement:

```
bid |= (IOCCSR_KEY | IOCCSR_SELECT); /* see <sys/iocc.h> */
```

To perform I/O to IOCC space, one uses the following macros:

```
IOCC_ATT(bid, iocc_addr)  /* to place Bus ID in a segment reg */
BUSIO_PUTL(long_val, iocc_addr)    /* to write four bytes */
BUSIO_GETL(long_val)               /* to read four bytes */
BUSIO_PUTS(short_val, iocc_addr)   /* to write two bytes */
BUSIO_PUTC(char_val, iocc_addr)    /* to write one byte */
IOCC_DET(iocc_addr)
```

To perform I/O to bus I/O space (adapter registers):

```
BUSIO_ATT(bid, io_addr)  /* to place Bus ID in a segment reg */
BUSIO_PUTLX(long_val,  io_addr)  /* to write four bytes */
BUSIO_GETLX(long_val)            /* to read four bytes */
BUSIO_PUTSX(short_val, io_addr)  /* to write two bytes */
BUSIO_PUTCX(char_val,  io_addr)  /* to write one byte */
BUSIO_DET(io_addr)
```

To perform I/O to bus memory space (adapter RAM):

```
BUSMEM_ATT(bid, mem_addr)
BUS_PUTLX(long_val, io_addr)    /* to write four bytes */
BUS_GETLX(long_val)             /* to read four bytes */
BUS_PUTSX(short_val, io_addr)   /* to write two bytes */
BUS_PUTCX(char_val, io_addr)    /* to write one byte */
BUSMEM_DET(io_addr)
```

The macros suffixed with an 'X' have exception handlers built into them. The return value is nonzero if an I/O exception occurs during the data transfer. I/O to IOCC control space does not generate an I/O exception if there is an error. To verify that data has been written correctly, you need to read it after it is written.

It is possible to use macros that lack exception handlers, but if you don't provide your own exception handler, then an I/O exception would cause the system's default handler to halt the system. There are other macros defined in the header file **/usr/include/sys/ioacc.h**.

To perform byte-reversed I/O reads:

`BUS_GETLRX( long *`*`ioaddr`*`, long *`*`data`*` )`

Reads the specified long value (*data*) from the supplied bus memory or bus I/O address (*ioaddr*) in byte-reversed format with built-in exception catching. The return value is 0 for success. Otherwise, an error (exception) occurred during the transfer.

The IOCC (I/O controller) on the system automatically converts 32-bit transfers from **LITTLE ENDIAN** format on the device into **BIG ENDIAN** format. Therefore, this macro undoes this conversion. Use this macro when the device or the data on the device is stored in **BIG ENDIAN** format instead of the usual **LITTLE ENDIAN** format found on most Micro Channel adapters.

`BUS_GETSRX( short *`*`ioaddr`*`, short *`*`data`*` )`

Reads the specified short value (*data*) from the supplied bus memory or bus I/O address (*ioaddr*) in byte-reversed format with built-in exception catching. The return value is 0 for success. Otherwise, an error (exception) occurred during the transfer.

The IOCC (I/O controller) on the system automatically converts 16-bit transfers from **LITTLE ENDIAN** format on the device in **BIG ENDIAN** format. Therefore, this macro undoes this conversion. Use this macro when the device or the data on the device is stored in **BIG ENDIAN** format instead of the usual **LITTLE ENDIAN** format found on most Micro Channel adapters.

To perform byte-reversed I/O writes:

`BUS_PUTLRX( long *ioaddr, long data )`

Writes the specified long value (*data*) to the supplied bus memory or bus I/O address (*ioaddr*) in byte-reversed format with built-in exception catching. The return value is 0 for success. Otherwise, an error (exception) occurred during the transfer.

The IOCC (I/O controller) on the system automatically converts 32-bit transfers from **BIG ENDIAN** format on the system in **LITTLE ENDIAN** format as seen by the device. Therefore, this macro undoes this conversion. Use this macro when the device or the data on the device is stored in **BIG ENDIAN** format instead of the usual **LITTLE ENDIAN** format found on most Micro Channel adapters.

`BUS_PUTSRX( short *ioaddr, short data )`

Writes the specified short value (*data*) to the supplied bus memory or bus I/O address (*ioaddr*) in byte-reversed format with built-in exception catching. The return value is 0 for success. Otherwise, an error (exception) occurred during the transfer.

The IOCC (I/O controller) on the system automatically converts 16-bit transfers from BIG ENDIAN format on the system in **LITTLE ENDIAN** format as seen by the device. Therefore, this macro undoes this conversion. Use this macro when the device or the data on the device is stored in **BIG ENDIAN** format instead of the usual **LITTLE ENDIAN** format found on most Micro Channel adapters.

To read the specified data from bus memory or bus I/O address space:

`BUS_GETSTRX( char *ioaddr, char *daddr, int count )`

Copies the number of bytes specified by the *count* parameter from either bus memory or the bus I/O address, which starts at the address specified by the *ioaddr* variable to memory starting at the location specified by the *daddr* parameter with built-in exception catching. The return value is 0 for success. Otherwise, an error (exception) occurred during the transfer.

To write the specified data to bus memory or bus I/O address space:

`BUS_PUTSTRX( char *ioaddr, char *saddr, int count )`

Copies the number of bytes specified by the *count* parameter from memory specified by the *saddr* parameter to bus memory or the bus I/O address (starting at the address specified by the *ioaddr* parameter*)* with built-in exception catching. The return value is 0 for success. Otherwise, an error (exception) occurred during the transfer.

Programmed I/O (PIO) Error Recovery Considerations for Micro Channel Adapters

All I/O operations typically include provisions to handle detectable errors. Except for Programmable Option Select (POS) accesses, programmed I/O (PIO) operations can contain a variety of synchronous errors. Because of this, device drivers must support synchronous I/O-error exception handling.

For PIO macros with built-in exception catching, a nonzero return code indicates that an exception or error occurred during the operation. If the operation is a read operation, the contents of the destination data area are not valid. If the return code is nonzero, it will have one of the exception values defined in the **/usr/include/sys/except.h** file.

An exception value of **EXCEPT_IO** indicates that the exception is an error caused by PIO and should be handled. If the return code is nonzero and not **EXCEPT_IO**, another error (usually a programming error) has occurred. The programmer coding this PIO operation should include an **assert** to stop the system when this error occurs and provide a dump. The operation can be retried up to the number of times specified by the **PIO_RETRY_COUNT** value. If the operation is still unsuccessful, handle it as a permanent error. The usual action is to return **EIO** to the caller of the device driver.

POS operations, unlike other PIO operations, cannot detect synchronous I/O errors. Device driver programmers should implement an algorithm that reads back the data after a read or write operation to see if a data error occurred. If the data resulting from this subsequent read operation does not match the results of the previous read or write operation, retry the POS operation. This procedure can be repeated up to the number of times specified by the **PIO_RETRY_COUNT** value. If the operation is still unsuccessful, handle this problem as a permanent error.

Direct Memory Access (DMA) on Micro Channel

The following subjects are discussed:

DMA Channels and How They are Assigned on Micro Channel

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. “Understanding DMA Arbitration Level Assignment” on page 2-24 discusses arbitration level assignments for bus master and slave DMA operations for the Micro Channel bus.

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

The system 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 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.

An adapter’s configure method reads the assigned DMA channel or channels from the ODM database for the specific adapter being configured and puts these channels in a device-dependent structure (DDS) 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 POS registers. This action properly configures the adapter.

Understanding DMA Arbitration-Level Assignment

All Micro Channel DMA devices typically support a Programmable Option Select (POS) function to select one of several possible arbitration levels on which to request DMA support. Any arbitration level can be defined to support either DMA slave or DMA master operations. The system supports 15 arbitration levels. Arbitration level 0 is the highest priority and arbitration level 14 the lowest.

The possible DMA arbitration levels supported by a specific adapter are defined by an attribute of type A in the Predefined Device database object for the adapter. This attribute has both a list of specified values and a default value for the adapter. Adapter arbitration levels are not a shared resource. To discover the arbitration levels a device supports, refer to the appropriate hardware reference manual.

Device Methods and DMA Arbitration Levels

When it detects the adapter during system boot, the bus Configure method calls the adapter Define method. This method creates a customized device entry for the adapter in the Customized Devices Object Class database. The bus Configure method then assigns the Micro Channel resources for all adapters, while attempting to avoid conflicts with resources that cannot be shared.

The bus Configure method tries to assign the adapter the same value the adapter had when last configured on the system. If the adapter was not previously configured, the Configure method attempts to use the default value. If these values conflict with other adapter resources, the configure method tries every possible value listed for the adapter until it finds one that resolves the conflict or exhausts all possible choices. If the conflict cannot be resolved, the conflicting adapter in the highest numbered slot is not configured. The arbitration level selected for the device is then stored in the adapter Customized Device Attribute object.

When the Configuration manager later calls the adapter Configure method, this method loads the device driver and builds a device-dependent structure (DDS). The Configure method uses attribute information from the Customized Device object class to specify the adapter initialization parameters. The DDS should contain the arbitration level assigned to the adapter as well as the bus identifier and slot number to which the adapter is attached. The Configure method then invokes the device driver **ddconfig** entry point for device initialization. The device driver must then save the device-dependent data received at initialization time and program the adapter POS registers with the values found in the DDS.

Allocating System Resources for DMA Support

The allocation and initialization of system support for resources such as DMA and interrupts should be delayed, if possible, until the first open request. This delay frees resources for other adapters until the adapter is needed. After the first open request for a device is received, allocate the system resources required by the adapter. For DMA support, this involves having the device driver call the **d_init** kernel service and specify the arbitration level, the bus identifier, and a *flags* parameter value that indicate the type of DMA service requested. Possible values for the *flags* parameter are defined in the **/usr/include/sys/dma.h** file and should be set to **MICRO_CHANNEL_DMA** for bus master support or **MICRO_CHANNEL_DMA+DMA_SLAVE** for DMA slave operations.

If the requested arbitration level is not allocated, the **d_init** kernel service returns the channel identifier (ID) *(channel_id* parameter) assigned to the adapter. This value is used by all other DMA services. If an error is returned, the open request for the device is rendered unsuccessful with the **EIO** return code. When it receives the last close operation, the device driver supporting the adapter should free the system resources allocated for the adapter. The DMA resource can be freed by calling the **d_clear** kernel service and passing it the channel ID of the DMA resource to be freed.

Direct Memory Access (DMA) Slave Operations

DMA slave transfer, the simpler method of device DMA, requires that the device driver:

- Ensure that the area in system memory is pinned.
- Obtain a cross-memory descriptor for the memory region by using the **xm_attach** kernel service.
- Set up the DMA channel by calling the **d_slave** kernel service.
- Initiate the DMA through programmed I/O to the adapter.
- Receive the operation-completion interrupt from the adapter.
- Call the **d_complete** service to flush the I/O controller buffers and check for transfer errors.

The **d_slave** kernel service supports setting up the DMA channel specified by the *channel_id* parameter for a data transfer between either the adapter and system memory or the adapter and another memory-mapped adapter on the bus. The **d_slave** kernel service must be used to set up a DMA channel for each DMA transfer.

If the DMA transfer consists of moving data to or from system memory, the memory area must be in contiguous virtual address space and must be pinned. The memory area must also have an associated cross-memory descriptor (obtained by the **xmattach** service). It is accessible at data transfer and interrupt time.

Make sure the data area in system memory is not accessed from the time the transfer is mapped using the **d_slave** service until after the transfer has completed and the **d_complete** service is called.

Setting up the DMA Channel

For a typical DMA transfer, the **d_slave** kernel service hides the system memory pages involved in the transfer to avoid inconsistency in data caused by the processor and I/O controller data-caching mechanisms. Any process attempting to access the memory region targeted by the transfer sleeps until the transfer is complete and the **d_complete** kernel service has been called.

Warning: Interrupt-handler access to memory regions involved in the transfer (or hidden by the processor), before the **d_complete** operation resolves can cause the system to crash.

Hiding pages can adversely affect memory regions within a global buffer area or a kernel buffer area. When accessing memory here, specify the **DMA_NOHIDE** flag in the call to the **d_slave** kernel service to avoid hiding pages and improve path length. However, this mode should be used only when transferring data to a device. Otherwise, the caching effects of the processor and I/O controller may cause data inconsistency in adjoining memory areas.

If the **DMA_NOHIDE** flag is used when transferring data from a device to system memory, assess the alignment of the affected data areas to ensure that caching does not result in data inconsistency. The data buffer involved in this case should be aligned on a cache line boundary and must end on a cache line boundary. The length of the data buffer, is typically 128 bytes.

System Memory Transfers

When calling the **d_slave** service for a transfer involving system memory, the following are required:

- The *channel_id* parameter
- The effective address of the system memory buffer
- The length of the transfer
- A cross-memory descriptor for the system memory buffer
- A *flags* parameter value

The *flags* parameter defines the DMA direction and type. For a transfer from system memory to a device, set the *flags* parameter to 0 for a normal transfer or use **DMA_NOHIDE** to avoid hiding the pages involved in the transfer. For a transfer from the device to system memory, make the *flags* parameter specify **DMA_READ**. The **d_slave** kernel service uses the I/O controller tag table to map the contiguous virtual memory region to the underlying non-contiguous physical pages. This service also flushes any processor data-cache lines that may contain data for the virtual memory range involved in the transfer.

Adapter-to-Adapter Transfers

When calling the **d_slave** service for a transfer from an adapter to another memory-mapped adapter on the I/O bus, the device driver requires:

- The channel ID (*channel_id* parameter) associated with the device performing the DMA
- The I/O bus-memory address of the other adapter
- The length of the transfer
- A *flags* parameter value

The *flags* parameter defines the DMA direction and type. For a transfer from another memory-mapped bus region to this device, set the *flag* parameter to **BUS_DMA**. For a transfer from this device to some other memory-mapped adapter area on the I/O bus, make the *flags* parameter specify **DMA_READ+BUS_DMA**.

After the device acknowledges the completion of the transfer, or indicates an error through an interrupt, make the device driver call the **d_complete** kernel service to flush any buffers from the I/O controller and check for errors. A call to this service also makes available any pages hidden by the transfer.

Steps for DMA Slave I/O for Micro Channel

As an example, if the driver is to transfer data from a user buffer to a DMA slave adapter, some steps for the transfer could be:

Steps for DMA Slave I/O for Micro Channel	
Module	**What Device Driver Could Do**
xyzopen	Call **d_init** to allocate and initialize DMA channel.
xyzread or xyzwrite	Call **d_slave** to map pages of I/O space or RAM to Tags. Perform PIO write to adapter register to start DMA transfer. Check for PIO exception and report any error.
xyzintr	(Interrupt received from adapter when DMA transfer complete.) Call **d_complete** to unhide pages and check for errors detected by IOCC.
xyzclose	Call **d_clear** to free DMA channel.

DMA Bus Master Operations

Direct memory access (DMA) bus master support is more complex to program than DMA slave operations, but provides a more flexible method of device DMA. During DMA bus master operations, the adapter controls the bus and generates the addresses for the data transfer. These addresses generated by the adapter are I/O bus-memory addresses. The system has a 4GB Micro Channel bus address range that is separate from the 4GB system-memory address range. Consequently, the DMA bus master transfers involving system-memory must have their addresses translated from I/O bus-memory addresses to system memory addresses.

Steps for DMA Master I/O for Micro Channel

As an example, if the driver is to transfer data from a user buffer to a DMA master adapter, some steps for the transfer could be:

Steps for DMA Master I/O for Micro Channel	
Module	**What Device Driver Could Do**
xyzopen	Call **d_init** to allocate and initialize DMA channel.
xyzread or xyzwrite	Call **d_master** to map pages of I/O space or RAM to TCWs. Perform PIO write to adapter register to start DMA transfer. Check for PIO exception and report any error.
xyzintr	(Interrupt received from adapter when DMA transfer complete.) Perform PIO read of adapter register to check DMA status. Check for PIO exception and report any error. Call **d_complete** to unhide pages and check for errors detected by IOCC.
xyzclose	Call **d_clear** to free DMA channel.

Managing the Bus Memory DMA Transfer-Window Region

The device driver that supports a bus master DMA device must manage this bus-memory window region into system memory as a limited resource since this region is supported by a set number of TCEs that can be used for the adapter. That is, when I/O requests are accompanied by data transfers, space must be allocated out of this fixed window region to system memory to map the system memory involved in the transfer about to be requested. Many device drivers may find it necessary to hold up and queue I/O requests until a sufficient space in this bus-memory region becomes available to map the requested transfer.

Note: When the device driver allocates a portion of its bus-memory window to map a system memory region, the device driver must preserve the same page offset (low 12 bits) of the system memory address in the newly allocated bus-memory address.

Mapping DMA Bus Master Transfers

To learn more about two essential techniques for mapping bus master DMA transfers, consult the following information:

- Short-Term Buffer Mapping
- Long-Term Buffer Mapping

Short-Term Buffer Mapping

The short-term method is preferable when transferring data to and from caller-supplied data areas that can be anywhere in system memory. The **d_master** kernel service is called to set up the association between the specified system memory buffer and the region of the adapter's bus-memory allocation used for DMA requests. The adapter is told to start the DMA request, using the bus-memory addresses that map to the correct area of system memory.

To perform a short-term transfer, the device driver must accomplish these tasks:

- Allocate the required space from the bus-memory region used for the DMA transfer window. If space is insufficient, either break up the transfer or queue the request until space is available.
- Ensure that the specified area in system memory is pinned.
- Obtain a cross-memory descriptor for the memory region.
- Set up the DMA mapping by calling the **d_master** kernel service.
- Initiate the DMA operation by using programmed I/O to the adapter specifying the bus-memory address used to map the transfer.
- Receive the operation completion interrupt from the adapter.
- Call the **d_complete** kernel service to flush the I/O controllers and check for transfer errors.

If the DMA transfer consists of moving data to or from system memory, the memory area must be in contiguous virtual address space and must be pinned. The memory area must also have an associated cross-memory descriptor (obtained by using the **xmattach** service). It is accessible at data transfer and interrupt time. Additionally, ensure the data area in system memory is not accessed from the time the transfer is mapped using the **d_complete** service until after the transfer has completed and the **d_complete** service is called.

Both cache-consistent and cache-inconsistent hardware platforms are supported by the DMA services. During a typical DMA transfer on a cache inconsistent platform, the **d_master** service hides the system memory pages involved in the transfer. This avoids data consistency problems due to the processor and I/O controller data-caching mechanisms. Any process attempting to access the memory region involved in the transfer will sleep (as if waiting on the resolution of a page fault) until the transfer has completed and the **d_complete** service has been called.

Warning: Interrupt handler access to this area before the **d_complete** operation has been issued will cause the system to crash.

Hiding of pages has an adverse effect within a global buffer area or a carefully managed kernel buffer area. If the memory region is in one of these regions, a **DMA_WRITE_ONLY** flag may be specified on the call to the **d_master** service. This avoids the hiding of pages and enforces read-only mode of the memory with respect to the bus master device. This mode can be used only when transferring data to a device. Otherwise, data inconsistency in adjoining memory areas could occur due to caching effects of the processor and I/O

controller. “Long-Term Buffer Mapping” on page 2-29 contains information concerning the data caching effects.

Note: On cache-consistent platforms, such as the PowerPC, pages are not hidden during DMA operations since data consistency is managed by the hardware. The **DMA_WRITE_ONLY** flag still enforces the read-only memory mode, but this has no effect on page hiding.

Calling the **d_master** service for a transfer involving system memory requires:

- The channel ID (*channel_id* parameter)
- The effective address of the system memory buffer
- The length of the transfer
- A cross-memory descriptor for the system memory buffer
- The associated bus-memory address
- The *flags* parameter

The *flags* parameter defines the DMA direction and type. For a transfer from system memory to a device, set this parameter to 0 for a normal transfer. And set the parameter to **DMA_WRITE_ONLY** to avoid the hiding of pages involved in the transfer and to enforce read-only mode by the bus master. For a transfer from the device to system memory, the flags should specify **DMA_READ**.

The **d_master** kernel service uses the I/O-controller TCE table to map the contiguous virtual memory region to the underlying noncontiguous physical pages. The **d_master** kernel service provides the association between I/O bus-memory address and physical memory address. This service also flushes any processor data-cache lines and I/O-controller prefetch buffers that may contain data for the virtual memory range involved in the transfer.

Long-Term Buffer Mapping

Long-term buffer mapping involves mapping the memory region for DMA transfers with the **d_master** kernel service. Use the **d_master** kernel service when repeatedly accessing the memory region for multiple DMA transfers with the device. This method is used to avoid the high costs of continuously remapping the memory region to the DMA transfer window under the following conditions:

- The memory region is permanently pinned. Remapping is required if the region is ever unpinned.
- The memory user is not in user space.
- The **DMA_WRITE_ONLY** or **DMA_NOHIDE** flag is specified on the **d_master** service.
- Data consistency is appropriately managed using the **d_cflush** and **d_move** kernel services.

Note: No special consideration is required on the cache-consistent PowerPC platform. Data consistency is managed at the hardware level on this platform. The **DMA_WRITE_ONLY** and **DMA_NOHIDE** flags are recognized, but have no effect on page hiding since pages are never hidden on the PowerPC platform.

Long-term buffer mapping is typically used when a memory buffer area is used as a command area. It is also used when a large number of data transfers come from a common, permanently pinned buffer pool. Long-term mapping requires that the bus-memory DMA window region assigned to the device be large enough to map this buffer area.

Many devices can use both short-term and long-term mapping in cases where a portion of the DMA window is mapped to a fixed area of system memory used for device command and response control blocks. The remaining area must be used for data transfers that are mapped for each request. Long-term mapping of response areas is supported by using the **d_master** service with either the **DMA_NOHIDE** or **DMA_WRITE_ONLY** flag. The **DMA_WRITE_ONLY** flag enforces a read-only mode with respect to the bus master device when accessing the system memory mapped in this manner. The **DMA_NOHIDE** flag allows for long-term mapping of read or write areas.

Due to the processor data cache and the I/O-controller caching functions, long-term buffer mapping requires special consideration on the system. The effects of these caches are hidden when using short-term mapping. This is because the **d_master** service performs the required processor cache flushes and the I/O controller cache becomes unusable and makes the system memory involved in the transfer inaccessible to the processor until the transfer is complete (**d_complete** time).

Two Methods of Long-Term Buffer Mapping

Using long-term mapping adds two new considerations to the use of long-term buffer mapping:

- Modification of data in the mapped memory areas after they have been mapped
- Shared concurrent access by both the processor and the device to the system memory area

There are two methods for handling long-term mapping based on the size and organization of the long-term mapped area. The first method involves aligning any data area in a transfer on cache-line boundaries and managing the caches using the **d_cflush** kernel service. To determine the data-cache-line size and I/O-controller cache size, refer to the appropriate technical reference for your device. When using a long-term mapped area for both commands (DMA write operations) and for responses (DMA read operations), the two cache areas must be in separate virtual memory areas. This is required to avoid data consistency problems due to the processor and the device concurrently updating the same area in memory.

Given that the DMA write and DMA receive areas are in different areas of memory, and that each DMA write area is in its own area, the sequence of events for this type of DMA operation is:

1. Select the data to be used for the DMA read or write operation.
2. Mark the data area as allocated. For example, remove it from the free list.
3. If this is a DMA write operation, update the data in the memory area chosen.
4. Call the **d_cflush** kernel service to flush and make unusable the processor and I/O caches.
5. Inform the device that a command or response area is available (giving the address).

Many of these operations can occur synchronously with a device DMA operation or a device interrupt response.

- Receive an interrupt from the device indicating an operation has been performed using the data area.
- Call the **d_complete** service, specifying which data area to check for errors and then ensure that the I/O controller buffers have been flushed to memory.
- Mark the data area as free to be used by another device transaction.

This first method of DMA long-term buffer mapping is highly useful for devices that use bus master DMA for command and status control blocks transfers. It is also useful when many relatively small (typically less than 1.5KB) data transfers occur, and it is more efficient to copy the data to a long-term mapped buffer than it is to map the short transfers. In particular, this method of DMA support is well-suited for devices using the subsystem control block (SCB) architecture for processor-to-device communications.

A long term mapped buffer pool of command control blocks (each on 64-byte boundaries) can be created, pinned, and permanently mapped to the device DMA window along with a buffer pool of status blocks for responses from the device. To provide for long-term mapping of response areas, use the **DMA_NOHIDE** flag. It should be specified on the single **d_master** call used to map the response area to allow the affected pages to be accessible to the processor while mapped for device DMA. When long-term mapping response areas, the processor cache for the response area must be make unusable before reading from the area. To make the affected cache line unusable, a **vm_cflush** call should be made before the DMA operation is started.

This approach is also commonly used by communications device handlers that perform device DMA directly into and out of **mbuf** structures. For DMA write operations, small **mbuf** structures typically come from a common-pinned pool. Therefore, they must be mapped without page hiding to avoid system crashes that could be caused by hiding a page in the common **mbuf** pool. These small **mbuf** structures are normally aligned on cache line boundaries and are an integral number of cache lines in size. For DMA read operations, cluster or page-sized **mbuf** structures are used for device DMA because the size of the incoming transfer is not known ahead of time.

These page-size **mbuf** structures are page-aligned and can be used in the **DMA_NOHIDE** mode for increased performance and when:

- Transferring data from unaligned or user data areas where many small transactions are normally known to occur.
- The cost of pinning the user or kernel data area and then mapping the transfers is higher than performing a memory-to-memory copy from the user or unaligned buffer to a permanently pinned and mapped buffer in the long-term mapped area.

The second method of long-term buffer mapping for DMA support is required where the DMA transfer areas cannot be blocked into cache-line boundaries (64B or 128B areas) of system memory or when extremely small (2B or 4B, for example) DMA transactions occur between the device and system memory. Communications devices use this method where the device and the processor share a long-term mapped buffer in system memory. For example, in a communications adapter, there may be pointers to communications input, output, and exception queues. Some of these queues are updated by the processor and others by the device itself. In this example, there may be cases where the processor updates a certain 4 bytes in the memory and the device updates an adjacent 4 bytes asynchronously.

Even if DMA write and read areas are sufficiently separated, flushing a 128-byte processor cache line to modify only a few bytes is inefficient. To solve this problem with both data inconsistency and inefficiency, the **d_move** kernel service uses a special I/O controller function that allows the processor to access data in system memory by bypassing the processor caches. The service uses the same I/O controller caches as the device itself does when transferring data. The **d_move** service allows the device driver writer to move small amounts of data to or from this shared memory area without having to be concerned with data consistency due to the processor and I/O caches. This method of data transfer should be restricted to instances where the previous method of cache management is unusable (due to data organization, for instance) or where the transfers are so small and frequent that the cache management approach is inefficient.

Processor write Operation to Shared Data Area

The sequence of steps in a processor write operation to shared data area is:

1. Move data from a nonshared buffer to the shared memory area using the **d_move** service. If this operation returns an **EINVAL** return code, use the **xmemin** or **xmemout** cross-memory services to move the data.
2. Call the **d_complete** kernel service with the address of the shared memory area to ensure that no errors occurred unless the **d_move** service returned an **EINVAL** code.
3. The device performs a DMA operation on the data when indicated by a command or some asynchronous event.
4. Call the **d_complete** service in response to a device-error interrupt to obtain any DMA error conditions and enable a retry of the DMA operation, if necessary.

Processor read Operation from Shared Data

The sequence of steps in a processor read operation from shared data is:

1. Call the **d_complete** service with the address of the shared memory area to ensure that no errors occurred during the DMA transfer before accessing the data or in response to a device-error interrupt.
2. Access the data by using the **d_move** kernel service from the shared data area to some other data area. If the **EINVAL** value is returned, use the **xmemin** or **xmemout** cross-memory service to move the data.
3. Call the **d_complete** service with the address of the shared memory area to ensure that no errors occurred unless the **d_move** service returns the **EINVAL** value.

The **d_move** kernel service moves data at I/O speeds, as opposed to normal processor speeds. This service can become inefficient for moving data approaching the size of a processor cache-line. The **d_move** kernel service is available on system models using software-managed caches. However, if models supporting transparent caches are introduced, the **d_move** kernel service is not required and returns an **EINVAL** return code. All users of this kernel service must check for an **EINVAL** return code. If an **EINVAL** return code is returned, the user should perform a normal data move. Doing so allows the device driver to operate without change, both on models with software-managed caches, and models where software cache management is not required.

Peer-to-Peer Bus Master DMA

DMA slave operations support transferring data from one device to another device's memory-mapped I/O area. Bus master operations can also be programmed to support bus master DMA directly between two devices. Like the **d_slave** service, the **d_master** service also supports the **BUS_DMA** flag, which indicates that the DMA transfer for the specified arbitration level is between the device and another device on the bus and not system memory. A drawback of bus-to-bus DMA master transfers is that all transfers on that arbitration level are directed either to I/O bus-memory or to system memory. Bus-to-bus transfers cannot be intermixed with bus-to-memory transfers on the same arbitration level since the current I/O controller supports only transfer mapping by arbitration level instead of mapping by translation-control word.

When an arbitration level is dedicated to one specific device and programmed I/O is used for commands and status to the device, the **d_master** service can be used to allow a direct adapter-to-adapter transfer of data. However, if multiple devices are concurrently supported by an adapter or if bus master DMA is used for command and status transmission to the device, multiple arbitration levels are required to support both bus-to-bus and bus-to-system memory transfers by the device. When programming bus-to-bus transfers, the I/O controller currently monitors the bus for errors during the transfer. The **d_complete** service can be called after the transfer to pick up any associated error information and enable the arbitration level again.

Alignment Issues for DMA on Micro Channel

As described under "Hiding DMA Data" in "Understanding DMA Transfers" in the chapter on kernel services in *AIX Kernel Extensions and Device Support Programming Concepts*, pages covering buffers for use by DMA services may be hidden unless DMA_NOHIDE is specified. If the buffer is supplied by a user, and if the buffer is not page aligned, then there is a chance that nearby data could be on the same page as the buffer being hidden. If a user routine attempts access to such data, its context will sleep until the page is unhidden. If the driver does not unhide the page, then such user routines would sleep indefinitely.

Any buffers that a driver allocates for DMA transfers may preferably be allocated from the kernel heap (via **xmalloc**) so that the buffer can be page aligned: this lessens the chance of nearby data being inadvertently hidden, and simplifies the coordination of data transfer with the adapter. Also, fewer pages may be used.

If a DMA buffer is to be shared between the system processor (the driver) and a DMA master adapter, then it would be registered with the flag DMA_NOHIDE. If a DMA buffer is to be so shared, or shared between two DMA master adapters (where page hiding is irrelevant), then there is a chance of data corruption due to race conditions (see Chapter 5, Sychronization). An I/O controller reads and writes data in chunks which are the size of its data cache line. If a DMA transfer starts at an address that is not cache-aligned, then nearby data is inadvertently accessed. A race condition can be avoided by synchronizing the access, or by ensuring that DMA accesses are cache-aligned and thereby non-overlapping. A buffer can be cache aligned with the kernel services **d_align** or **d_roundup**.

Here is an example of a call-side routine that sets up DMA master transfers on a Micro Channel adapter. Please note that the buffers are allocated by the driver itself and they are accessed by the system and the adapter, so they are not hidden (DMA_NOHIDE) from the system.

```
int dma_init()
{
 caddr_t busptr; /* points to bus address space */
 int rc; /* followed by other necessary declarations */

scb.ccbp = (t_ccb *) xmalloc(PAGESIZE, PGSHIFT, pinned_heap);
 if (scb.ccbp == NULL)
 {      DTRC(0x20, 0x43, (ulong) scb.ccbp);/* custom trace macro */
        return(ENOMEM);
 }

 /* clear the buffer for use */
 bzero(scb.ccbp, PAGESIZE);

 /* SCB - System Control Block /* you would define your own */
 * TSB - Termination Status Block
 * Separate regions must be set up for transferring data for DMA
 * reads and DMA writes. These separate regions should each consist
 * of an integral number of cache lines. The Command Control
 * Block (CCB) starts on the page boundary, which is also a cache
 * line boundary. The CCB and TSB must reside in separate 128-byte
 * regions for cache consistency.
 */
 scb.ccb_size = d_roundup(sizeof(t_ccb)); /* size of ccb struct */

 /*
 * The algorithm used by d_roundup() is :
 * outlength = (inlength + cache_line_size-1) & (cache_line_size-1);
 * where inlength is the length in bytes of the structure to be mapped.
 *
 * So the next available cache line boundary is at ccbp + ccb_size
 */
 scb.tsbp = (uint)scb.ccbp + (uint)scb.ccb_size;

 /* need the size of tsb also for flushing */
 scb.tsb_size = d_roundup(sizeof(t_tsb)); /* size of tsb struct*/

 /* the amount of data that is actually used is:(should be 256 bytes)*/
 scb.ctl_size = (uint)scb.tsb_size + (uint)scb.ccb_size;

 /* The ccb is mapped to the beginning of the bus memory window
 * configured for this device. The tsb follows.
 */
 scb.bus_ccbp = dds.dds_hdw.dma_bus_mem; /* device dep. struct */
 scb.bus_tsbp = (uint)scb.bus_ccbp + (uint)scb.ccb_size;

 /* initialize the ccb before mapping */
.....
 /* set up cross memory descriptor for the ccb and tsb */
 scb.ctl_xmem.aspace_id = XMEM_INVAL;
 rc = xmattach((char *)scb.ccbp, scb.ctl_size, &scb.ctl_xmem, SYS_ADSPACE);
```

```
 if (rc == XMEM_FAIL)
 {      DTRC(0x20, 0x43, (ulong) rc);
        xmfree(scb.ccbp, pinned_heap);
        return(ENOMEM);
 }

 /* map the control area to the first page of bus dma memory space */
 d_master(scb.chan_id, DMA_READ|DMA_NOHIDE, (caddr_t)scb.ccbp,
        scb.ctl_size, &scb.ctl_xmem, scb.bus_ccbp);

 /* attach to the bus segment */
 busptr = BUSIO_ATT(dds.dds_hdw.bus_id,(caddr_t)dds.dds_hdw.bus_io_addr);

 /* write address of the ccb to the command port, it never changes */
 port.command = (caddr_t) scb.bus_ccbp;
/* pio_retry() logs temporary failure & after 3rd fail gives up */
if(BUS_PUTLRX((busptr + HLZ1_COMMAND), port.command))
        rc = pio_retry(rc, PUTL, addr, port.command);

if (port.pio_fatal) /* check for a permanent PIO exception */
 {      DTRC(0x20, 0x45, (ulong) rc);
        xmdetach(&scb.ctl_xmem);
        xmfree(scb.ccbp, pinned_heap);
        BUSIO_DET(busptr);
        return(EIO); /* must read this reg to continue */
 }

 /* set the dma busmem address for the input buffer mapping */
 dma.indmap = dds.dds_hdw.dma_bus_mem + PAGESIZE;
 dma.window_size = dds.dds_hdw.dma_bus_length - PAGESIZE;
dma.enddmap = dds.dds_hdw.dma_bus_mem + dds.dds_hdw.dma_bus_length;

 /* initialize mapping control variables */
 dma.last_inbuf = NULL;
 dma.last_outbuf = NULL;

 /* enable the device as a busmaster, enable interrupts,clear IV on read*/
 port.sub_ctrl = .... /* device flags */;
 if(rc = BUS_PUTCX((busptr + HLZ1_SUB_CTRL), port.sub_ctrl))
        rc = pio_retry(rc, PUTC, addr, port.sub_ctrl);

 if (port.pio_fatal) /* check for a permanent PIO exception */
 {      DTRC(0x20, 0x46, (ulong) rc);
        xmdetach(&scb.ctl_xmem);
        xmfree(scb.ccbp, pinned_heap);
        rc = EIO; /* must read this reg to continue */
 }
 BUSIO_DET(busptr);
 return(rc); /* error already logged if PIO exception */

} /* end dma_init() */
```

Chapter 3. Interrupts

This chapter describes the issues associated with writing an interrupt handler for an adapter on AIX Version 4.1. It contains the following sections:

Overview

Adapters on any bus can generate interrupts to the host processor. Each interrupt is associated with a particular level, sometimes called an Interrupt Request Level (IRQ). For example, one adapter can generate interrupts on IRQ2, although another can generate interrupts on IRQ3. Assignment of interrupt levels to adapters is done by one of the following methods:

- Setting POS registers on the adapter during device initialization. Some buses do not support this.
- Manually setting the interrupt level via hardware switches.

Interrupt levels on certain buses, like the Micro Channel or PCI, can be shared. This means that more than one adapter can generate interrupts at the same level. In this case, each adapter must provide a register that can be interrogated to determine if the current interrupt is due to this particular adapter. This enables the software to determine which adapter actually generated the interrupt. Having each adapter on a separate interrupt level usually gives better performance than interrupt-sharing on a particular level.

Each adapter also provides a procedure, which software must follow, that resets an interrupt indication. This must be done before any more interrupts are generated by the adapter.

Device drivers usually provide an interrupt handler to handle these situations. An interrupt handler is a function that is called by the AIX kernel whenever an interrupt occurs on a given IRQ level. The interrupt handler must first determine whether the interrupt was caused by the adapter this driver is managing. If not, the handler exits immediately indicating no action taken. If so, the interrupt handler performs whatever processing is needed to deal with the interrupt, resets the interrupt in the adapter, and returns to the AIX kernel.

Interrupt handlers, like most of the AIX kernel, can be preempted. They run with some interrupts enabled. When a device driver configures itself, it specifies the priority of interrupts from its associated adapter. When an interrupt occurs, only interrupts from devices at that priority level and below are disabled. Higher priority interrupts can still occur. Interrupt handlers for low priority devices (such as, a printer) can be preempted if an interrupt occurs on a high priority device (such as, an unbuffered high-speed communications link).

The AIX operating system provides routines so that interrupts at higher levels can be disabled and enabled during the operation of the interrupt handler. Use these routines with caution so that the higher priority interrupts can be serviced.

Interrupt Hardware Support

The processor recognizes the following types of hardware interrupts:

- SYSTEM RESET
- MACHINE CHECK
- DATA STORAGE
- INSTRUCTION STORAGE
- EXTERNAL
- ALIGNMENT
- PROGRAM
- FLOATING-POINT UNAVAILABLE
- DECREMENTER (available on PowerPC only)
- SYSTEM CALL
- TRACE
- FLOATING-POINT ASSIST

The hardware interrupts in the preceding list are the only way that normal instruction execution can be interrupted. Thus all interrupt signals must be converted to one of these to interrupt the processor. An interrupt vector for each type of hardware interrupt is loaded in low memory and is executed upon receipt of the interrupt. The first two types of interrupts, SYSTEM RESET and MACHINE CHECK, may occur at any time regardless of the state of the interrupt mechanism and are processed immediately. The other interrupts are handled in the order listed from highest to lowest priority.

The hardware interrupt type EXTERNAL is the interrupt into which all bus level interrupts are multiplexed. This is the type of interrupt that a device driver will be required to handle.

For the PowerPC architecture, the Machine State Register (MSR) and the Save Registers (SRR0 and SRR1) are used to process external interrupts. The MSR register contains an enable bit which is used to enable or disable external interrupts. The SRR registers are used to save the processor state at the time of the interrupt. For more details on the these registers and the other types of hardware interrupts, see *PowerPC Architecture*.

The hardware interrupt mechanism between the processor and the device is bus architecture dependent. For Micro Channel architecture, the interrupt controller is embedded in the I/O Channel Controller (IOCC). For PowerPC based platforms with PCI and ISA buses, a System I/O Bridge chip (SIO) and a proprietary PCI Bridge/Controller are used to relay the interrupt signal to the processor as an external interrupt.

Interrupt Levels

The mechanism by which the processor knows where the interrupt originated depends on a software abstraction of the interrupt source referred to as *processor interrupt level*. (*Processor interrupt level* is also called *software interrupt level*, or just *interrupt level*.) The interrupt level corresponds to a distinct hardware interrupt level present on a device. For example, each bus interrupt level that originates on the PCI bus is converted into a unique software interrupt level that the processor understands and uses to call the proper interrupt handler for that interrupt. This design isolates the software from the information the hardware presents to indicate the interrupt.

The number of interrupt levels that a processor recognizes is architecture dependent. The POWER machines (RS1, RSC, and RS2) support 64 interrupt levels, and the PowerPC machines can support up to 8000 interrupt levels. An interrupt must map into one of these levels to be correctly serviced; that is, each hardware interrupt level must be assigned to one of the processor interrupt levels. The details of this mapping process is described in “Interrupt-Level Mapping,” on page 3-6.

Because individual adapters interrupt at various hardware levels of their own depending on what type of bus they are on, there is one more configuration issue to consider. Some bus

implementations support assigning hardware interrupt levels 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 hardware levels and initialization data to reflect this assignment.

For example, on the Micro Channel bus, there are 16 bus interrupt levels. A typical Micro Channel adapter has several bus interrupt levels from which to choose. Before setting which bus interrupt level to use, the adapter's configuration method calls the **busresolve** routine. The **busresolve** routine returns a bus interrupt level that does not conflict with any other adapters on the bus. The bus interrupt level returned by **busresolve** is written to the ODM database. The device driver open routine can refer to the ODM database to register the interrupt handler at the correct level.

However, some buses, like the ISA bus, do not support programmable assignment of bus interrupt levels. The assignment of these bus interrupt levels is usually hardwired or selected by a jumper on the adapter. In this case, the Predefined Attributes PdAt entries of the ODM database must be set properly by the user to reflect the manual settings (or the manual settings changed to match the ODM database), because there is no way for the values to be changed by software. Driver developers should remember this important difference about ISA bus configuration.

Attention: Do not change the Predefined Attributes (PdAt) object class by removing information that was shipped with the base operating system.

On some buses, such as the PCI and MCA buses, interrupt level sharing is supported. To ensure that **busresolve** handles these levels correctly, the ODM stanza for interrupt level must have the type field set to `"I"`. Non-shareable interrupts (all those on the ISA bus, for example) are declared with type set to `"N"`. Shared interrupt levels on the MCA buses *must* all request the same priority, but this is not required for the PCI or ISA bus.

The following sample ODM entry declares an interrupt level to be shareable by setting the attribute type to `"I"`:

```
PdAt:
        uniquetype = "adapter/pci/sample"
        attribute = "intr_level"
        deflt = "15"
        values = "15"
        width = ""
        type = "I"
        generic = "D"
        rep = "nl"
        nls_index = 3
```

Here is an example of a non-shareable interrupt (`type = "N"`):

```
PdAt:
        uniquetype = "adapter/isa/sample"
        attribute = "intr_level"
        deflt = "9"
        values = "5, 7, 9"
        width = ""
        type = "N"
        generic = "DU"
        rep = "nl"
        nls_index = 4
```

Chapter 6, "Device Configuration Methods," on page 6-1 has more information on the ODM database and adapter device attributes.

The following table summarizes the interrupt level properties for different types of buses.

Bus	Interrupt Sensitivity	Interrupt Levels Shareable?	Programmable Interrupt Levels?	Available Interrupt Levels
MCA	Level	Yes, but must have same priority.	Yes	1 – 16
PCI	Level	Yes	Yes	15
ISA	Edge	Basically, no. Yes, if level-triggered.	No	5,7,9,11,14,15 *

*IRQ 15 is used by PCI or ISA, but not both.

In some documentation, the distinction between hardware interrupt levels and processor interrupt levels is not made, because there is a one-to-one mapping between the two. Usually, just the term interrupt level is used.

Interrupt Priorities

Each processor interrupt level (see “Interrupt Levels,” on page 3-2) also has a interrupt priority associated with it. The following is a list of all the interrupt priorities (sorted so the interrupt priorities with higher priority appear first):

- INTMAX
- INTCLASS0
- INTCLASS1
- INTCLASS2
- INTCLASS3
- INTOFFL0
- INTOFFL1
- INTOFFL2
- INTOFFL3
- INTIODONE
- INTBASE

These interrupt priorities represent the order in which the kernel enables external interrupts to be processed. (The **sys/m_intr.h** file shows the numeric value associated with each priority. The higher priorities have lower numeric values.)

Although the physical mechanisms vary for the different architectures, a basic rule for all architectures is that the interrupt priority must be of a higher priority than the current processor priority for the interrupt to be serviced. For example, if the processor is currently servicing an interrupt of priority INTCLASS3, any off-level interrupts will not be serviced until the INTCLASS3 interrupt handling is finished, and the processor priority is lowered.

The choice of what priority to run your device driver interrupts is based on two criteria:

- The maximum interrupt latency requirements
- The interrupt execution time of the device driver

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. The interrupt execution time is the number of machine cycles required by the device driver to service the interrupt.

A device with a short interrupt latency time must have a short interrupt service time. In other words, a device that *loses* data if not serviced quickly must have a higher priority interrupt level. This in turn requires that it spends less time in the interrupt handler. The following list contains general guidelines for interrupt service times:

Interrupt Priority	Service Time
INTMAX	All interrupts disabled
INTCLASS0	Less than 200 cycles

INTCLASS1	Greater than 200 but less than 400 cycles
INTCLASS2	Greater than 400 but less than 600 cycles
INTCLASS3	Greater than 600 but less than 800 cycles
INTOFFL0	Less than 1500 cycles (off-level priority)
INTOFFL1	Greater than 1500 but less than 2500 cycles (off-level priority)
INTOFFL2	Greater than 2500 but less than 5000 cycles (off-level priority)
INTOFFL3	5000 cycles or greater (off-level priority)
INTIODONE	I/O completion processing (lowest off-level priority)
INTBASE	All interrupts enabled

Note: To find out your interrupt service time, you can put a trace hook with a time stamp into both the entry and the exit points of your interrupt handler. The resulting trace tells you the cumulative time that was spent in the handler. Divide this time by the cycle speed of your system to get the number of cycles used.

The INTOFFL*n* (where *n* represents an integer) 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 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 action in turn enables interrupt handlers to run as fast as possible by avoiding interrupt-processing delays and device overrun conditions.

Note: This routine must be part of the bottom half of the device driver when one is present.

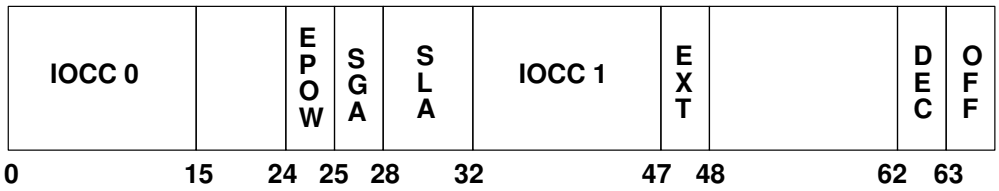
Once an interrupt priority has been determined, the information is written to the system via an ODM entry. The following sample entry assigns an interrupt priority INTCLASS2, which has the value 3, to a device:

```
uniquetype = ”adapter/mca/hscsi”
attribute = ”intr_priority”
deflt = ”3”
values = ”3”
width = ””
type = ”P”
generic = ”D”
rep = ”nl”
nls_index = 5
```

Interrupt-Level Mapping

The mapping of interrupt source to a specific processor interrupt level is dependent upon the hardware architecture. This section briefly describes the mappings for the POWER, POWER2, and PowerPC architectures and, also, discusses software interrupt priority.

The POWER architecture has a simple static mapping that directs the interrupt sources to one of the 64 available processor levels. Since there are at most 2 IOCCs, each bus has its 16 hardware levels directed to 32 of the existing processor levels and the rest are used for other interrupt sources. The POWER Interrupt Level Mapping figure shows the details of the processor level mapping scheme.



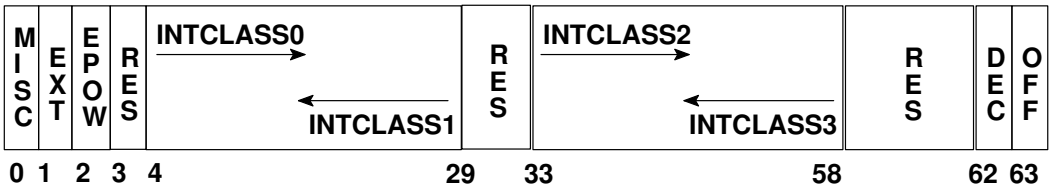
LEVEL	ASSIGNED
0–15	IOCC 0
24	Early Power-Off Warning (EPOW)
25	SGA – integrated graphics adapter
28	SLA0
29	SLA1
30	SLA2
31	SLA3
32–47	IOCC 1
48	External Check
62	Decrementer
63	Off-level hardware assist

POWER Interrupt Level Mapping

The interrupts are grouped into priorities by using a mask built and maintained by the interrupt subsystem called the External Interrupt Mask (EIM). This mask is changed whenever a new priority is set, thereby enabling the correct processor levels.

Although the POWER2 architecture also has 64 processor levels, it does not have the priority flexibility of the POWER architecture. There is a direct relationship of processor level and priority. Of the 64 processor levels, the hardware presents interrupts from the most favored (level 0) to the least favored (level 63). Thus, interrupt levels must be dynamically assigned based on priority.

The interrupt level allocation algorithm groups the processor levels into ranges corresponding to software priorities, and assigns them as needed. This ensures that higher priority interrupts are serviced before those of lesser priority. In addition to these dynamically allocated levels, there are other interrupt levels that are pre-allocated to specific sources. These can be seen in the POWER2 Interrupt Level Mapping figure.

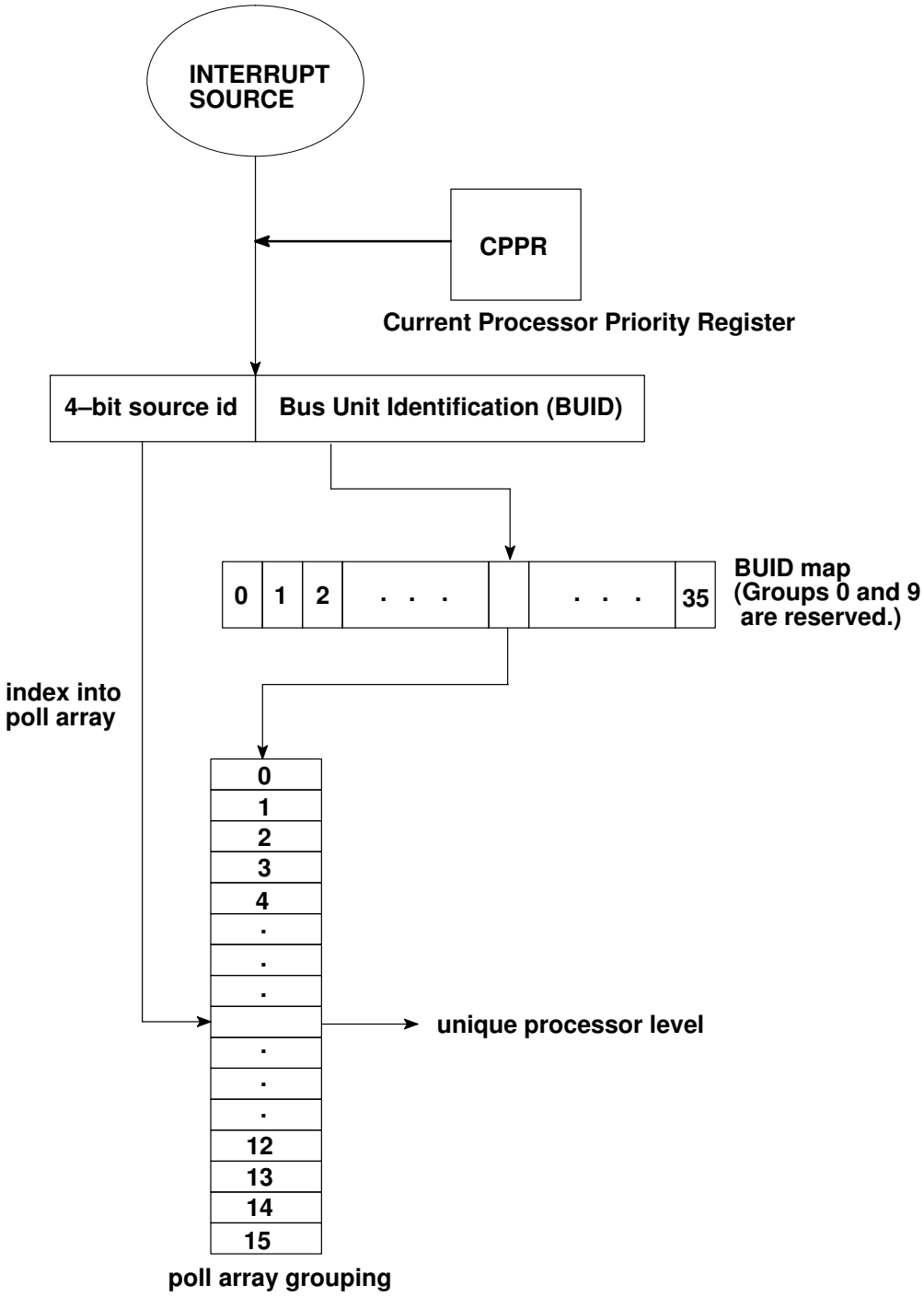


LEVEL	ASSIGNED
0	Miscellaneous IOCC interrupts
1	External Check
2	Early Power-Off Warning (EPOW)
3	Reserved
4–28	INTCLASS0, INTCLASS1 dynamically assigned interrupts
29–32	Reserved
33–57	INTCLASS2, INTCLASS3 dynamically assigned interrupts
58–61	Reserved
62	Decrementer
63	Off-level hardware assist

POWER2 Interrupt Level Mapping

The PowerPC interrupt logic is very different from the previous architectures. Each hardware level interrupt has a priority associated with it and will only interrupt on processors if this priority is more favored than the current processor priority. The processor priority is set by software for each processor.

Attached to each interrupt source is the Bus Unit Identifier (BUID) and a 4-bit code that indicates the source on the specific Bus Unit Controller (BUC). These two values are used by the interrupt subsystem to determine the interrupt level. On the PowerPC architecture, there are a possible 8000 levels, of which 160 are currently being used. These levels are assigned to groups that contain 16 interrupt levels. The poll groups are dynamically assigned to a BUID through the buid_map when the interrupt handler is registered. This mapping is shown in the PowerPC Interrupt Level Mapping figure.



PowerPC Level Mapping

Interrupt Handling

When an external interrupt is first detected, the system immediately calls the external interrupt first-level interrupt handler (FLIH), which queries the hardware registers. At this point on POWER and POWER2 machines, the interrupt is directly serviced. On PowerPC machines, however, the FLIH will enqueue the interrupt based on level and priority. This raises a flag indicating that there is pending work to be done and it will be serviced later, thus the PowerPC essentially enqueues all interrupts.

The kernel detects queued interrupts at various key times, such as when enabling to a less-favored priority from a more favored one. Once a queued interrupt is detected and the processor is executing at (or about to be enabled to) a priority that lets the pending interrupt be serviced, the current machine state is saved, and interrupt processing is started.

Then the kernel begins calling the interrupt handlers that are registered at the specified level. Because interrupt levels can be shared on certain buses, the adapter which caused the interrupt is not necessarily known at this stage. The order in which the kernel calls interrupt handlers at a certain level is the order in which they were initially registered. This ordering does not change as long as the interrupts are registered. Thus, there is no way to "steal" interrupts from a previously loaded handler at the same level and priority.

Once your second-level interrupt handler (SLIH) is called, it must determine whether the associated adapter caused the interrupt. If the interrupt was caused by the adapter, the interrupt handler does its necessary work, possibly schedules more off-level work, and returns INTR_SUCC. If the interrupt was not caused by the adapter, INTR_FAIL is returned, and the kernel calls the next handler in the list.

Early Power-Off Warning Interrupt

Special interrupt subsystem support is provided to handle loss of power. Some machines detect when 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 from loss of power.

For example, the file system 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** (same as **INTMAX**) and a bus type of **BUS_NONE**. 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 short, since the amount of time between the early power-off warning and actual power loss is usually short (a few milliseconds). (This timing is hardware-dependent.) Only critical data devices such as disks need to register an EPOW handler.

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

Calling a registered EPOW interrupt handler is different from calling other interrupt handlers registered by the **i_init** service. There are three conditions under which registered EPOW handlers are called:

EPOW_SUSPEND
Calling 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
Calling 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
Calling 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. The following situations 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 called again 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.

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.

BUS Interrupts

In general, the ISA bus has edge-triggered interrupts. The PCI and MCA buses use level-sensitive interrupt lines. The 8259 Programmable Interrupt Controller (PIC) has level-sensitivity program control for each of the IRQ lines. At startup, the 8259 IRQs are initialized for edge sensitivity. The MP Interrupt Controller (MPIC) IRQs are initialized to level sensitivity. When an interrupt handler is registered using **i_init()**, flags in the **intr** structure determine whether or not the trigger mode of the associated IRQ is changed. These flags (**INTR_EDGE**, **INTR_LEVEL**, and **INTR_POLARITY**) are in the `flags` field of the **intr** structure.

The `flags` field indicates whether the device driver allows sharing of an interrupt level and whether the interrupt handler is MPSAFE. In addition, this field contains other flags used by interrupt handlers that understand and need to know when an Early Power-Off Warning interrupt has occurred. The following is a table of values:

Flag Name	Value	Description
INTR_NOT_SHARED	0x0001	Interrupt level cannot be shared.
EPOW_SUSPEND	0x0002	Power loss in progress.
EPOW_RESUME	0x0004	Wall power has resumed.
EPOW_BATTERY	0x0008	Running on battery power.
INTR_MPSAFE	0x0010	MP-safe interrupt handler.
INTR_EDGE	0x0020	Interrupt level has edge-triggered semantics. Edge-triggered interrupts cannot be shared.
INTR_LEVEL	0x0040	Interrupt level has level-triggered semantics. Level-triggered interrupts can be shared.
INTR_POLARITY	0x0080	Interrupt polarity 0–Active High or Positive Edge 1–Active Low or Negative Edge
I_SCHED	0x8000	If set, already scheduled.

The bus type of the IRQ (PCI, ISA, etc.) does not determine the trigger mode of the IRQ. It must be specified explicitly in the **intr** structure. Since ISA and PCI buses have default triggering semantics, **i_init()** attempts to know the trigger mode of the IRQ being connected to when the **INTR_LEVEL** and the **INTR_EDGE** flags of the intr structure are not set. A request to register a PCI interrupt results in the **INTR_LEVEL** flag being set; an ISA interrupt request results in the **INTR_EDGE** flag being set; and an MPIC interrupt request results in the **INTR_POLARITY** flag being set. The **INTR_POLARITY** flag must be set to 0 for MCA-based and PCI-based devices. Bus interrupt levels on the ISA bus are usually not shared, while PCI bus interrupt levels are. The **INTR_NOT_SHARED** flag is set by device drivers that cannot share interrupts.

Because the 8259 is subject to priority inversion (a lower interrupt holding off a more favored interrupt), interrupts are EOIed as soon as possible in the external first-level interrupt handler (FLIH). For any interrupt level that is shared and level sensitive, special action is required before the interrupt is issued an EOI. The IRQ line is masked in the 8259 before the EOI is issued. After the interrupt is serviced, the IRQ line is unmasked. If the level is shared and the interrupt line is still being asserted, another interrupt occurs.

If a bus level is shared by multiple interrupting devices, each device can have a different priority for ISA and PCI sources only. If this is the case, the processor level associated with the shared, multi-priority bus level is checked for service at the lowest priority in the list. The linked list associated with the processor level is maintained in descending priority order. However, the interrupt logic adjusts the priority of the system before calling each handler in the list. For example, consider a case where device driver A registers to interrupt as a shared handler on IRQ15 using priority level INTCLASS0. Device driver B registers for the same bus level, but it wants to be serviced at priority INTCLASS3. In this example, the entire linked list associated with processor level is checked for service at INTCLASS3. The list is kept in such an order that driver A is called before driver B, thus maintaining the priority in some sense.

Note: It is recommended that interrupt sources on the same interrupt line use the same priority.

Interrupt Management Kernel Services

The following list contains interrupt management kernel services.

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.

Note: A system assert will occur if this service is called for an undefined interrupt handler.

i_disable Raises the interrupt priority to the specified level, thus disabling all interrupt levels at a less-favored interrupt priority.

i_enable Restores the interrupt priority to a less-favored interrupt priority, thus enabling all interrupt levels of a higher priority.

i_mask Disables the specified bus interrupt level.

i_unmask Enables the specified bus interrupt level.

i_sched Schedules an off-level interrupt handler to be executed.

The **i_reset** kernel service, which was used to reset a bus interrupt level in releases before AIX Version 3.2.5, no longer needs to be explicitly called. Its function has been moved to the first-level interrupt handler (FLIH), and thus interrupt levels are automatically reset.

Multiprocessor Interrupt Concerns

Ensuring proper synchronized access to the interrupt handlers is of major concern when writing handlers for a multiprocessor (MP) environment. AIX Version 4.1 provides two kernel services that enable you to make your interrupt handling MP-safe:

disable_lock Raises the interrupt priority, and locks a simple lock if necessary.

unlock_enable Unlocks a simple lock if necessary, and restores the interrupt priority.

Essentially, these kernel services should be used wherever **i_enable** and **i_disable** are normally used in a uniprocessor interrupt handler. The simple lock kernel services should *not* be called directly, use **disable_lock** and **unlock_enable** to ensure that a thread will never be interrupted while it holds a simple lock. Allowing a thread which holds a simple lock to be interrupted can deadlock the system.

In an MP environment it is not necessary for all the interrupt handlers to be MP-safe; however, this is a desired goal for increased performance. If an interrupt handler that is not MP-safe is registered, the allocated interrupt level is marked for funnelled operation, which is serviced by the master processor only. Additionally, all the other interrupt handlers at this level will be routed to the master processor. In other words, all interrupt handlers sharing the same level are either considered all MP-safe or all funnelled. Once all interrupt handlers that are funnelled are removed from a level (by calling **i_clear**), any remaining MP-safe handlers on that level will be allowed to run non-funnelled.

In addition, defining and removing interrupt handlers from the system must be done carefully. If the corresponding interrupt handler is executing on another processor when these services are called, the request cannot be performed until the handler exits. If the handler is spinning on a lock held by the calling thread, the system will deadlock. Therefore, ensure that any thread-interrupt lock which could block the corresponding interrupt handler is released before calling these kernel services.

Interrupts on PCMCIA Devices

Interrupts on PCMCIA devices are triggered by level mode. The IRQ level can be shared between the PCMCIA bus and the PCMCIA devices.

It is preferable that a PCMCIA device driver not assume interrupts are triggered by level mode. A device driver should query **CardServices** by a call twice to the RequestIRQ function by setting or clearing the CSIRQLevel flag of the IRQInfo1 member in the requesting packet. If the call to RequestIRQ succeeds, a device driver knows the interrupt trigger mode (INTR_EDGE or INTR_LEVEL|INTR_POLOARITY) that is isued to call the **i_init( )** kernel service.

Chapter 4. Memory Management

This chapter discusses the various system calls typically utilized by kernel extensions and device drivers in the manipulation of kernel memory. These services include:

- Memory Allocation Services, on page 4-1 (allocate and free kernel memory)
- Memory Pinning Services, on page 4-3 (pin and unpin kernel memory)
- Memory Access Services, on page 4-5 (transfer data between user memory and kernel memory)
- Virtual Memory Management Services, on page 4-6 (manage virtual memory)
- Cross-Memory Services, on page 4-11 (perform cross-memory transfers)

This chapter discusses only the most commonly used kernel memory services available to device driver developers.

Memory Allocation Services

The following are common memory allocation services:

- **xmalloc**
- **xmfree**
- **init_heap**

xmalloc

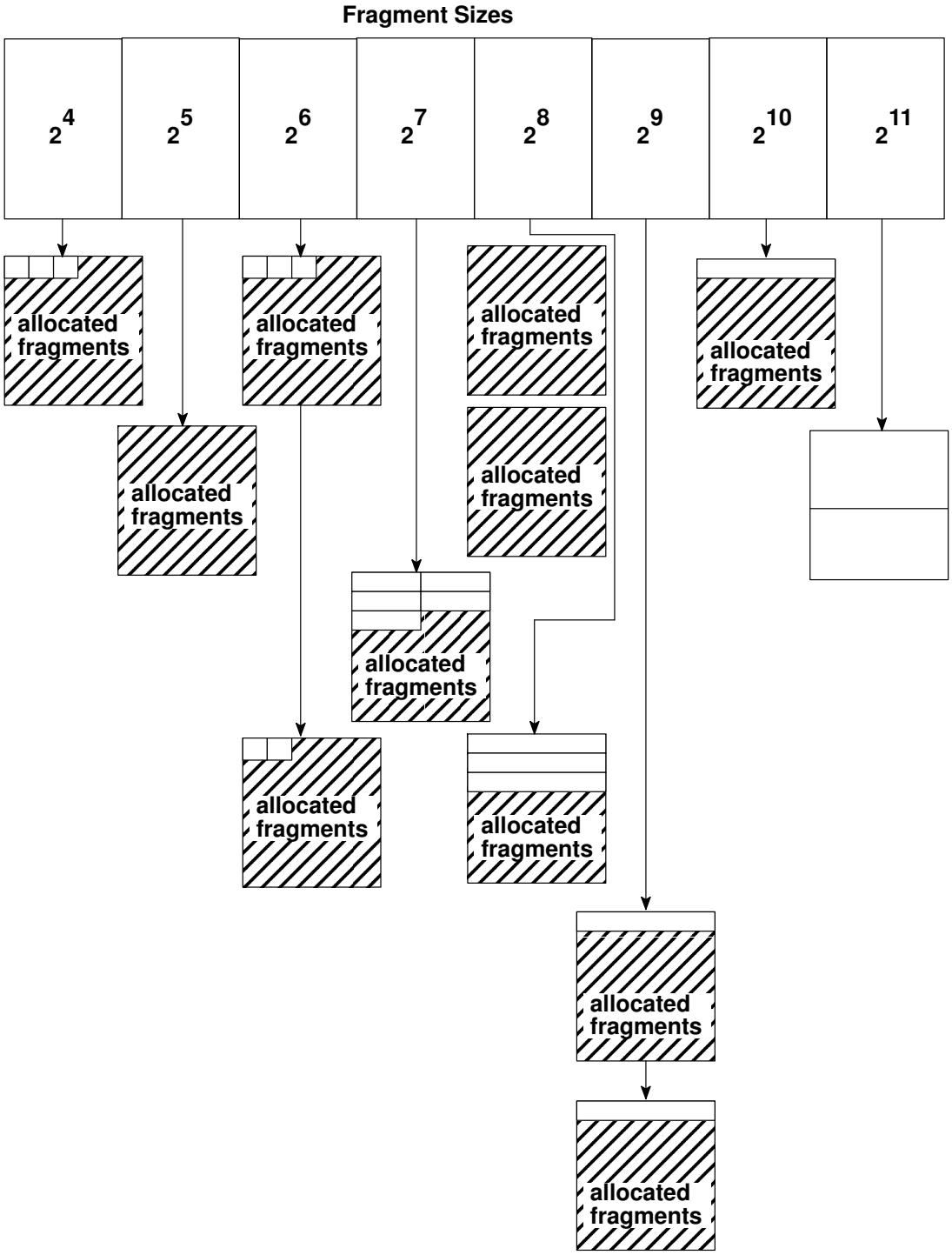
The **xmalloc** kernel service allocates an area of memory from either the kernel heap or the pinned kernel heap. Memory should be allocated from the pinned heap if it is intended to always remain pinned or remain pinned for a long period of time. If the allocated memory can be paged out, it should be requested from the kernel heap. Any unpinned memory can be pinned with the **pin** and **unpin** system calls at a later time if necessary.

The memory area returned by this service can be allocated on a boundary that is a power of 2 bytes from 16 bytes up to a page boundary of 4096 bytes (16-byte boundary, 32-byte boundary, and so on up to a 4096-byte boundary.)

For requests less than one page, **xmalloc** rounds up the request to the next higher power of 2. This implies that a request of just more than half a page is rounded up to one page. **xmalloc** also allocates a minimum of 16 bytes at a time due to the allocation algorithm it utilizes.

The **xmalloc** kernel service allocates requests differently based on the size of the request. For requests of half a page or less (requests greater than half a page are rounded up to one page), a vectored array is kept of elements that represent powers of 2 up to half a page. Each element is an anchor to a list of allocated pages that are individually divided into fragments. These fragments are equal to the size represented by the anchor array element. Once a page is used up (all fragments are allocated), a new page is then grabbed and divided and placed onto the appropriate list. This algorithm allows for fast allocation of areas less than half a page because lists of fragments of the correct size have already been allocated and are immediately ready for use by the caller. Although this method may seem wasteful since it preallocates a whole page of fragments for each size, the worst case only requires 8 pages to be pre-allocated with all free fragments.

The following figure shows the layout of the array used to keep track of the pages that are divided into the various fragment sizes. Note that as long as a page contains a free fragment, it is kept on the linked list for its size. Once all fragments on a page are allocated, the page is no longer linked to the array. When one fragment comes free on a fully allocated page, that page is reinserted onto the list corresponding to its fragment size.



Xmalloc Array of Fragments Less Than 1 Page

For allocations of greater than half a page, **xmalloc** allocates enough pages to contain the requested allocation size.

xmalloc cannot be called with interrupts disabled.

xmfree

Essentially, the **xmfree** kernel service frees up memory areas allocated through the **xmalloc** kernel service. In the case of allocated regions one page or greater, **xmfree** merely frees up the allocated pages.

In the case of memory fragments less than one page, **xmfree** must first find the page to which the fragment belongs as it resides in the vectored array as described in the discussion of **xmalloc**. Once the fragment is freed, the page to which it belongs is also freed if the freed fragment completes an entire page of freed fragments.

xmfree cannot be called with interrupts disabled.

init_heap

The **init_heap** kernel service allows a device driver or kernel extension to set aside an area of memory as a heap for private use. This reserved area must be page aligned and may be a subset of another heap. The **xmalloc** and **xmfree** kernel services are used to allocate and deallocate memory from this private heap.

init_heap cannot be called with interrupts disabled.

Memory Pinning Services

The following are common memory pinning services:

- **ltpin**
- **pin**
- **pincode**
- **pinu**
- **ltunpin**
- **unpin**
- **unpincode**
- **unpinu**

ltpin

The **ltpin** kernel service enables device drivers and kernel extensions to perform long-term pinning of pages of kernel memory. This prevents them from being paged out. All pages touched by the specified address range are pinned, not just the range itself. If the defined range overlaps just slightly into one page, the entire page is pinned.

The **ltpin** service increments a pin count for each page. A page is not a candidate to be paged out until its pin count reaches zero, typically by later calls to the **ltunpin** service.

The major difference between this service and the short-term **pin** kernel service is that long-term pinned pages have their corresponding paging space allocation freed but short-term pinned pages retain their paging space allocation.

There is also a limit to the number of long-term pinned pages allowed in the system. The maximum number of long-term pinned pages is just under 32767 pages.

ltpin cannot be called with interrupts disabled.

pin

The **pin** kernel service enables device drivers and kernel extensions to perform short-term pinning of pages of kernel memory. This prevents them from being paged out. All pages touched by the specified address range are pinned, not just the range itself. Therefore, if the defined range overlaps just slightly into one page, that entire page is pinned.

The **pin** service increments a pin count for each page. A page is not a candidate to be paged out until its pin count reaches zero, typically by later calls to the **unpin** service.

There is also a limit to the number of short-term pinned pages allowed in the system. The maximum number of short term pinned pages is just under 32767 pages.

pin cannot be called with interrupts disabled.

pincode

The **pincode** kernel service supports long-term pinning of an entire object module's code and data. This service calculates the address and length of a module's code and data and calls the **ltpin** kernel service with these values to perform the actual pinning.

The **pincode** service increments a pin count for each page. A page is not a candidate to be paged out until its pin count reaches zero, typically by later calls to the **unpincode** service.

pincode cannot be called with interrupts disabled.

pinu

The **pinu** kernel service provides short-term pinning of memory in either user or kernel space. This routine calls the **pin** kernel service after performing an attach to the user's address space.

The **pincu** service increments a pin count for each page. A page is not a candidate to be paged out until its pin count reaches zero, typically by later calls to the **unpinu** service.

pinu cannot be called with interrupts disabled.

ltunpin

The **ltunpin** kernel service unpins the memory pages long-term pinned by the **ltpin** kernel service.

This service only decrements the pin count of one or more pages. A page is only a candidate to be paged out when its pin count reaches zero.

The **ltunpin** kernel service cannot be called with interrupts disabled because it may need to allocate paging space for the newly unpinned memory area.

unpin

The **unpin** kernel service unpins the memory pages short-term pinned by the **pin** kernel service.

This service only decrements the pin count of one or more pages. A page is only a candidate to be paged out when its pin count reaches zero.

The **unpin** kernel service can be called with interrupts disabled.

unpincode

The **unpincode** kernel service unpins a module's code and data that was pinned by the **pincode** kernel service. Once it has calculated the addresses of a module's code and data segments, **unpincode** calls the **ltunpin** kernel service to perform the actual unpinning.

This service only decrements the pin count of one or more pages. A page is only a candidate to be paged out when its pin count reaches zero.

unpinu

The **unpinu** kernel service unpins memory pinned by the **pinu** kernel service. As in the **pinu** kernel service, memory from either user or kernel space can be freed. This routine calls the **unpin** kernel service to perform the actual unpin.

This service only decrements the pin count of one or more pages. A page is only a candidate to be paged out when its pin count reaches zero.

This routine must be called with all interrupts disabled when unpinning memory in user space. This routine can only unpin memory in kernel space if interrupts are disabled.

Memory Access Services

The following are common memory access services:

- **copyin**
- **copyinstr**
- **copyout**
- **uiomove**

copyin

The **copyin** kernel service copies data from user memory to kernel memory with appropriate exception handling.

copyin cannot be called with interrupts disabled.

copyinstr

The **copyinstr** kernel service copies a string from user memory to kernel memory with appropriate exception handling. This service copies up to the number of bytes specified or until a NULL byte is reached.

copyinstr cannot be called with interrupts disabled.

copyout

The **copyout** kernel service copies data from kernel memory to user memory with appropriate exception handling.

copyout cannot be called with interrupts disabled.

uiomove

The **uiomove** kernel service copies data between kernel memory and either user or kernel memory as defined by a **uio** structure. When using cross-memory descriptors, **uiomove** calls either **xmemin** or **xmemout** for the actual transfer. Cross-memory transfers also require a valid `uio_xmem` pointer to an array of **xmem** structures.

uiomove cannot be called with interrupts disabled.

uwritec

Device drivers writing out one character at a time can use the **uwritec** kernel service. 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 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.

ureadc

Device drivers reading in one character at a time can use the **ureadc** kernel service. 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 operation 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.

Virtual Memory Management Services

Virtual memory describes the hierarchy of both main system memory and secondary storage such as hard disks. This two-tier structure allows actual physical memory to be divided and allocated to several processes at the same time. It also allows programs to address memory far larger than the actual size of physical system memory because secondary storage can be used as an extension of the system memory.

Virtual memory objects are often used by device drivers to represent data that can be in either system memory or on a secondary storage device. The virtual memory address space is divided into *segments*, which are actually 256MB contiguous areas of this address space. Process addressability is managed at the segment level. These segments can then be kept private to a specific process or can be shared among several processes. Segments themselves are further divided into *pages*, which are currently 4096 bytes.

There are several types of segments. A *working* segment is a segment that does not have corresponding permanent storage space. For example, the stack and data region for a process are mapped to a working segment because they exist only within the context of the existence of the process. Once there is not enough physical memory to hold a working segment's pages, some of these pages must be transferred, or *paged out*, to paging space, which is a temporary area of secondary storage (for example, hard disk). Once these pages are again needed by the process, they can then be *paged in*, or transferred back into system memory.

Persistent segments are mapped segments that have a corresponding permanent storage area in secondary storage. Files on disk that are mapped are examples of *mapped* segments. When a page of a persistent segment must be paged out, the actual page must be written to secondary storage if it has changed while being mapped. If the page has not changed, it is merely discarded and its place in physical memory is now open for another page to be paged in.

Client segments are persistent segments that map files remotely, such as over an NFS mount. When pages for these types of segments are paged out, they are written out over the network.

Virtual memory objects are often used by device drivers to do things such as mapping a file that exists on a hard disk in secondary storage.

The following is an example of the usual steps to create and manipulate a virtual memory object that maps in an NFS mounted file:

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

vmid_t      vmid;
vmhandle_t  demo_vm_handle;
int         type;
struct gnode *gn;
int         size;
int         num_vm_bufs;
int         rc;
extern int  demo_nfs_strategy();
caddr_t     buffer;

num_vm_bufs = 50;    /* 50 is arbitrary for the number of buf
                        structures to allocate */
                     /* demo_nfs_strategy() is the strategy
                        routine that handles the pageouts */
rc = vm_mount(D_REMOTE, demo_nfs_strategy, num_vm_bufs);
if (rc)
     return (rc);
          . . .


type = V_CLIENT;  /* going to work with a "client" segment */
/* gn is a pointer to the gnode of the file to be mapped */
/* size is the size in bytes of the file to be mapped */
if (rc = vms_create(&vmid, type, gn, size, 0, 0))
     return (rc);
/* vms_create() actually returns a segment ID in vmid */

demo_vm_handle = vm_handle(vmid, 0);
/* vm_handle() actually returns a segment register value */
    . . .

buffer = vm_att(demo_vm_handle, 0);
                  /* we're just picking 0 as the offset */
/* Note that the kernel will panic if it runs out of spare
   segment registers but there should be enough to go around.
   As a general rule, a device driver should limit itself to 2
   attaches at a time. */

/* The buffer can now be written to and read from */
           . . .

vm_det(buffer);
           . . .
if (rc = vms_delete(vmid))
   return (rc);
           . . .
vm_umount(D_REMOTE, demo_nfs_strategy);
```

The kernel services used to manage virtual memory generally have names beginning "vms_" (for example, **vms_create**, **vms_delete**, and **vms_iowait**) or names beginning "vm_" (for example, **vm_handle**, **vm_att**, and **vm_cflush**.)

vms_create

The **vms_create** kernel service creates a virtual memory object of a specific type, size, and limits.

The type parameter, which is an OR of bits, should always include V_CLIENT to indicate a client segment, and may include the V_INTRSEG option to indicate the segment is an interruptable segment.

vms_create cannot be called with interrupts disabled.

vms_delete

The **vms_delete** kernel service deletes a virtual memory object created through the **vms_create** kernel service. Even though this service completes asynchronously, notification of completion is given synchronously. This is because the allocated segment is not truly freed until all paging I/O has completed. Until the segment has been marked as freed, it remains accessible even though a successful return code may have been received.

vms_delete cannot be called with interrupts disabled.

vm_handle

The **vm_handle** kernel service creates a virtual memory handle for a virtual memory object, as created by **vms_create**, for use by the **vm_att** kernel service. The handle is created from a concatenation of the segment ID and a protection key.

vm_handle cannot be called with interrupts disabled.

vm_att

The **vm_att** kernel service maps a virtual memory object, created by the **vms_create** kernel service, by allocating a free segment register and returning a 32-bit address made up of the segment register value and the offset within that segment. This 32-bit address has the segment register number in the upper 4 bits and the offset in the lower 28 bits.

If the system has no segment registers to allocate, it will panic.

vm_att can be called with interrupts disabled.

vm_cflush

The **vm_cflush** kernel service flushes out to memory all processor instruction and data cache lines that contain the memory boundaries specified by the given address and range.

vm_cflush can be called with interrupts disabled.

vm_det

The **vm_det** kernel service essentially unmaps the virtual memory object originally mapped by the **vm_att** kernel service. It does so by releasing the segment register associated with the given virtual address of the memory object.

vm_det can be called with interrupts disabled.

vm_mount

The **vm_mount** kernel service allocates an entry in the paging device table (PDT) for a file system and also allocates the required **buf** structures for processing by the strategy routine.

vm_mount cannot be called with interrupts disabled.

vm_umount

The **vm_umount** kernel service removes an entry from the paging device table (PDT) for a file system and also deallocates the required **buf** structures that were previously allocated by the **vm_mount** kernel service. If paging activity has not yet ceased when **vm_umount** is called, the service waits until all I/O has completed before completely freeing all resources.

vm_umount cannot be called with interrupts disabled.

vm_move

The **vm_move** kernel service will transfer data from a virtual memory object, created by the **vms_create** kernel service, and a buffer pointed to by a **uio** structure.

vm_move cannot be called with interrupts disabled.

The following is an example of using **vm_move** on a permanent hard disk file:

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

vmid_t     vmid;
int        type;
int        size;
dev_t      dev_num;
int        rc;
struct uio *uiop;
      . . .

type = V_CLIENT;  /* client segment */
    /* dev_num is the devno of the block device on which
       the inode resides */
   /* size is the size in bytes of the file */
if (rc = vms_create(&vmid, type, dev_num, size, 0, 0))
   return (rc);
if (rc = vm_move(vmid, uiop->uio_offset, uiop->uio_resid,
    UIO_READ, uiop));
   return (rc);
if (rc = vms_delete(vmid));
   return (rc);
     . . .
```

vm_write

The **vm_write** kernel service initiates a page-out for all pages touched by a specified range in a virtual memory object. The memory range is defined by a beginning address and length. All pages touched by this range are marked for page-out.

A *force* parameter, applicable only to journaled segments, is provided to force a page-out on a page even though it may have been recently modified. Typically, if a page has been recently modified, it will most likely be modified again. By delaying page-outs on recently modified pages, time is saved by writing the page only after several modifications rather than after each modification. The *force* parameter overrides this process.

vm_write cannot be called with interrupts disabled.

vm_writep

The **vm_writep** kernel service is similar to the **vm_write** kernel service except that a page range, rather than a memory range, is specified for page-outs.

No *force* parameter is provided.

vm_writep cannot be called with interrupts disabled.

vms_iowait

The **vms_iowait** kernel service waits until all page-out I/O for a virtual memory object is complete.

vms_iowait cannot be called with interrupts disabled.

vm_release

The **vm_release** kernel service releases all pages touched by a specified range in a virtual memory object. In essence, an address range is defined by a beginning address and length. All pages touched by this range are freed.

vm_release cannot be called with interrupts disabled.

vm_releasep

The **vm_releasep** kernel service will release the specified pages in a virtual memory object. It is similar to the **vm_release** kernel service except that pages are specified rather than an address range.

vm_releasep cannot be called with interrupts disabled.

Example Using Virtual Memory Management Services

The following example shows typical use of the **vm_writep**, **vms_iowait**, and **vm_releasep** kernel services:

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

vmid_t     vmid;
int        type;
int        size;
dev_t      dev_num;
int        pfirst;
int        npages;
int        rc;
       . . .

type = V_CLIENT;  /* client segment */
  /* dev_num is the devno of the block device on which
     the inode resides */
  /* size is the size in bytes of the file */
 if (rc = vms_create(&vmid, type, dev_num, size, 0, 0))
    return (rc);
      . . .

 /* pfirst is the page number of the first page to pageout */
 /* npages is the number of pages we want to pageout */
if (rc = vm_writep(vmid, pfirst, npages));
   return (rc);
rc = vms_iowait(vmid);
vms_releasep(vmid, pfirst, npages);
if (rc)
   return (rc);
      . . .
```

Cross-Memory Services

The cross-memory kernel services enable device drivers to access user-mode data. This is often required by the interrupt handler section of device drivers or when performing DMA transfers to and from a user buffer.

The following are common cross-memory services:

- **xmattach**
- **xmdetach**
- **xmemin**
- **xmemout**
- **xmemdma**

The **xmattach** kernel service provides a descriptor to access the user-mode memory region while the **xmemin** and **xmemout** kernel services transfer data to and from this region. The **xmdetach** kernel service then deallocates resources used to access the user-mode region and prevents further accesses to that region.

The following example shows a typical transfer of user data to a kernel buffer using the **xmemin** kernel service:

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

caddr_t   user_buffer;
caddr_t   local_buffer;
uint      buffer_len;
struct xmem     dp;
int       rc;

bzero(&dp, sizeof(struct xmem));
/* aspace_id field must be initialized to XMEM_INVAL */
dp.aspace_id = XMEM_INVAL;

/* user_buffer should have the address of the user buffer */
/* user_buffer_len should have the length of the user_buffer */
if (rc = xmattach(user_buffer,user_buffer_len,&dp,USER_ADSPACE))
   return (rc);

/* allocate word-aligned kernel bufr to hold copy of user data */
local_buffer = xmalloc(user_buffer_len, 2, pinned_heap);
if (local_buffer == NULL)
   return (ENOMEM);
if (rc = xmemin(user_buffer, local_buffer, buffer_len, &dp))
   return(rc);
/* The data in local_buffer can now be sent to the device
   if desired (eg. through PIO) */
    . . .

xmfree(local_buffer, pinned_heap);
xmdetach(&dp);
```

xmattach

The **xmattach** kernel service gives a device driver access to a user buffer without having to execute under the process that initiated the I/O. It returns a cross-memory descriptor if the attach is successful. The **xmemin** and **xmemout** kernel services can then be used to transfer data to and from the attached user buffer.

xmattach cannot be called with interrupts disabled.

xmdetach

The **xmdetach** kernel service detaches a user buffer, previously attached by the **xmattach** kernel service, from a device driver. This prevents a device driver from further accesses to a user buffer.

xmdetach can be called with interrupts disabled.

xmemin

The **xmemin** kernel service transfers data from an attached user buffer to a kernel buffer. The device driver performing the transfer should have previously attached to the user buffer using the **xmattach** kernel service.

xmemin can be called with interrupts disabled.

xmemout

The **xmemout** kernel service transfers data from a kernel buffer to an attached user buffer. The device driver performing the transfer should have previously attached to the user buffer using the **xmattach** kernel service.

xmemout cannot be called with interrupts disabled.

xmemdma

The **xmemdma** kernel service prepares a page for DMA I/O or processes a page after DMA I/O is complete. Even though **xmemdma** can be called with interrupts disabled, the page being processed must be in memory and either pinned or in the pager I/O state.

The following flags are used when preparing the page for DMA I/O:

XMEM_HIDE
: Flushes the cache and invalidates the page if this is the first hide for the page. (On a PowerPC machine, this flag instructs the kernel service to return the calculated real address for the page and if the page is not read-only, set the modified bit.)

XMEM_ACC_CHK
: Checks the page protection bits for this page before the flush and hide.

XMEM_WRITE_ONLY
: When used with the XMEM_ACC_CHK flag, indicates that page access is read-only and DMA transfers from this page will only be outward.

The following flag processes a page after DMA I/O:

XMEM_UNHIDE
: Decrements the hide count on this page. If this is the last unhide and the page is not in the pager I/O state, processes waiting on the page are taken off the wait queue and the page's modified bit is set unless it was a read-only page. (On a PowerPC machine, this flag causes the kernel service to return 0.)

xmemdma can be called with interrupts disabled.

The following is an example of the use of the **xmemdma** kernel service to unhide a page after a DMA transfer using the **d_master** kernel service:

```
int          channel_id;
caddr_t      buf_addr;
int          buf_size;
struct xmem  xmem_buf;
caddr_t      dma_addr;
      . . .
/* We're assuming that the channel_id, buf_addr, buf_size,
   xmem_buf,and dma_addr have already been setup */
 d_master(channel_id, DMA_READ, buf_addr, buf_size, &xmem_buf,
          dma_addr);
 xmemdma(&xmem_buf, buf_addr, XMEM_UNHIDE);
```

Chapter 5. Synchronization and Serialization

It is often necessary for a device driver to synchronize its access to a resource that is shared with a device or with kernel routines executing in other contexts. Kernel routines in all contexts share the same kernel segment and all data structures within it. Drivers for many devices share the same I/O space and system bus. A device driver's routines all access the same device, so they must synchronize their access to that device.

To synchronize access with another device, either the driver may *poll* the device by repeatedly checking to see if the resource is available, or the device may notify the driver that the resource is available by sending an interrupt whose handler posts an *event* that the driver is waiting for.

If a driver routine accesses a resource that it shares with a kernel routine concurrently executing in another context, then both routines must address various concurrency issues. For example, you want to avoid the following conditions:

race conditions Multiple contexts access the same shared resource with different results depending on the order in which the accesses are made. (Accesses to a shared resource are serialized if they are coordinated so there is no race condition.)

deadlock Each context awaits a notification the other never sends.

livelock Each context polls a condition the other never sets. (This is sometimes called a type of deadlock.)

starvation Other contexts monopolize access to the shared resource.

A portion of a routine that accesses a shared resource, a routine's *critical section*, makes the access *atomic* relative to routines executing in other contexts whenever it synchronizes its own access to that resource with respect to any access attempts by other routines, so that all accesses are serialized (made one after the other). A routine synchronizes access with other routines either by using semaphores (such as locks) or by invoking *monitors*, which are routines that ensure atomic access.

A routine is *reentrant* if its critical sections make resource accesses that are atomic relative to the same routine executing in some other context. In other words, a routine is reentrant if it can safely execute in many contexts concurrently.

If a routine maintains state information somewhere (for example, in static or global data) so that a subsequent call to the same routine would access information modified by a previous call, then the routine is not reentrant: the state information is the shared resource, and that portion of the routine that accesses the information is that routine's critical section. If a routine enables a caller to access any resources that can be shared, such as by returning a pointer to static or global data, then the routine itself is not reentrant; it cannot safely execute in more than one context.

Recall that an interrupt handler routine must be reentrant because a device may generate an interrupt while another interrupt (possibly from the same device through another system processor) is being handled. In AIX, device driver routines on the call side also must be reentrant because a call-side driver routine executes in the context of a kernel thread whose execution can be preempted in favor of another thread of greater priority, which might be executing the same call-side driver routine.

For information on making interrupt-side routines reentrant, see Chapter 3, "Interrupts" on page 3-1.

Timer Services

A driver routine can poll a device actively by checking the shared resource's availability and then doing something else (keeping a system processor), or it can poll a device by checking and then relinquishing its system processor for a certain period of time (going to sleep). Because the time between resource availability checks with active polling depends on the speed of a system processor, and because that time period may not be compatible with device being polled, timer-based polling is more robust.

Each processor that AIX supports has a decrementor that periodically generates *timer interrupts* whose handler, among other things, can invoke routines specified to be called once a certain amount of time has passed. The kernel services and data structures associated with timer interrupt handler are *timers*. There are basically two kinds of timers: watchdog timers and real-time timers. A driver routine sets up a timer by specifying a *callback routine*, which is an interrupt-side routine that is to be invoked by the timer interrupt handler. The driver routine also specifies when to invoke the callback routine as either an absolute time (from January 1, 1970) or a time relative to when the timer is started.

Watchdog Timers

Driver routines typically use timers to check if device I/O requests had successfully completed. A *watchdog* timer may be satisfactory to poll a device, as it is fairly simple to use and has low overhead. However, it involves writing an interrupt-side callback routine that is little more than a timed "go to" routine: a watchdog timer handler receives no user defined parameters. Also, a watchdog timer can only be set to the nearest second.

Here is an example of how to set up a watchdog timer. The following code sections are written to augment the sample device driver shown in "Device Driver Overview" on page 1-1.

The callback routine, which is invoked any time the watchdog timer expires, executes on the interrupt-side; so it will have to be in the device driver's bottom half. In other words, the callback routine will have to be explicitly pinned in RAM.

Here is the bottom half, `xyz_bot.c`:

```
#include <sys/types.h> /* for dev_t and other types */
#include <sys/trchkid.h> /* for trace hook macros */
/************************************************************
 WATCHDOG TIMER CALLBACK ROUTINE
This callback routine is invoked every time a timer expires.
This executes in the context of decrementer interrupt handler.
************************************************************/
void watchdog_callback()
{
TRCHKL1T(HKWD_USER1, 0xb);
}
```

Since the callback routine has no parameters, it must access some shared resources to do anything useful. It may read an adapter's registers to poll a device, or it may see if some other resource is available for use. Portions of the callback routine that access such shared resources are critical sections shared among threads and interrupt handlers and require appropriate synchronization.

The driver's top half, called `xyz_top.c`, is the same as the sample driver in Chapter 1 with several additions.

Add the following to the declaration section:

```
#include <sys/malloc.h> /* for xmalloc() */

/* for watchdog timer support */
#include <sys/watchdog.h> /* for watchdog structure */
extern void watchdog_callback();
struct watchdog *watchp;
```

Note that the pointer to the watchdog timer structure is global and is therefore a shared resource requiring synchronization if the top half were to be reentrant. For simplicity, access to the pointer is not serialized in this example.

Add the following to the `xyzopen` entry point:

```
watchp = (struct watchdog *) xmalloc(sizeof(struct watchdog),
        0, pinned_heap);
if(watchp == (struct watchdog *) NULL)
{ setuerror(ENOMEM); /* failed, no space on pinned heap */
 return(-1); /* setuerror() used: there is no ublock access */
}

watchp->func = (void (*)(void *))watchdog_callback;
w_init(watchp);
```

Add the following to the `xyzclose` entry point:

```
w_stop(watchp);
w_clear(watchp);
xmfree(watchp, pinned_heap);
```

Add the following to the `xyzread` entry point:

```
watchp->restart = 3; /* 3 seconds */
w_start(watchp);
```

Add the following to the CFG_INIT portion of the `xyzconfig` entry point:

```
if((return_code = pincode(watchdog_callback)) != 0)
{ setuerror(return_code);
 return(-1);
}
```

And, add the following to the CFG_TERM portion of the `xyzconfig` entry point:

```
if((return_code = unpincode(watchdog_callback)) != 0)
{ setuerror(return_code);
 return(-1);
}
```

The corresponding stanzas of the makefile need to be changed:

```
xyz: xyz_top.o xyz_bot.o
        ld -e xyzconfig -o xyz -bI:$(KSYSLIST) -bI:$(SYSLIST)
xyz_top.o xyz_bot.o                    # have on one line

xyz_top.o: xyz_top.c
        cc -c -D_ALL_SOURCE -D_POSIX_SOURCE -D_KERNEL xyz_top.c

xyz_bot.o: xyz_bot.c
        cc -c -D_ALL_SOURCE -D_POSIX_SOURCE -D_KERNEL xyz_bot.c
```

The user program, `aprogram`, only needs to have "`sleep(5);`" placed after the call to **read** so that the timer can go off before the program terminates.

The following sample trace report shows that the callback routine (hookdata 0xB) executed a little less then three seconds after the call to xyzread (hookdata 0x9):

```
 trace   -j010 -l -l -s -a

 ID  PROCESS NAME   I SYSTEM CALL     ELAPSED   APPL    SYSCALL KERNEL
INTERRUPT

 001 trace   0.000000      trace   ON channel 0
 010 trace  20.121544      UNDEFINED trace   ID idx 0x2100 trace  id 0010
     hookword 10E0000 type 0E
     hookdata 0000 00000001 00630000 00000001 2FF97F1C 00000000
 010 trace  26.447790      UNDEFINED trace   ID idx 0x2154 trace  id 0010
     hookword 10E0000 type 0E
     hookdata 0000 00000001 00630000 00000003 2FF97F1C 00000000
 010 trace  26.447795      UNDEFINED trace   ID idx 0x2170 trace  id 0010
     hookword 10A0000 type 0A
     hookdata 0000 00000005
 010 trace  63.647987      UNDEFINED trace   ID idx 0x22c4 trace  id 0010
     hookword 10E0000 type 0E
     hookdata 0000 00000007 00630000 00000003 00000000 00000000
 010 trace  63.648657      UNDEFINED trace   ID idx 0x22e0 trace  id 0010
     hookword 10E0000 type 0E
     hookdata 0000 00000009 00630000 2FF97DC0 00000000 00000000
 010 trace  66.057338      UNDEFINED trace   ID idx 0x2314 trace  id 0010
     hookword 10A0000 type 0A
     hookdata 0000 0000000B
 010 trace  70.649700      UNDEFINED trace   ID idx 0x2348 trace  id 0010
     hookword 10E0000 type 0E
     hookdata 0000 0000000A 00630000 2FF97DC0 00000000 00000000
 010 trace  70.649833      UNDEFINED trace   ID idx 0x2364 trace  id 0010
     hookword 10E0000 type 0E
     hookdata 0000 00000008 00630000 00000000 00000000 00000000
 002 trace  77.524587      trace   OFF channel 0
```

There are many similarities between routines using watchdog timers and those using real-time timers. This is shown in the following discussion of real-time timers.

Real-Time Timers

If the device requires setting a timer to within something finer than a second, or if the callback routine needs user defined parameters, then the driver will use real-time timers, which make use of Timer Request Blocks (TRBs) as defined in the file **/usr/include/sys/timer.h**.

The real-time kernel services are:

talloc Allocates a TRB.

tstart Submits the timer request (to place the TRB on a timer queue).

tstop Removes the timer request.

tfree Frees up resources allocated for the TRB.

Here is an example of a way to set up a real-time timer. The following code sections are written to augment the sample device driver shown in “Device Driver Overview” on page 1-1.

As with watchdog timers, the callback routine, specified to be called once the timer expires, executes on the interrupt-side and so will have to be in the device driver’s bottom half.

Here is the bottom half, `xyz_bot.c`:

```
#include <sys/types.h> /* for dev_t and other types */
#include <sys/errno.h> /* for errno declarations */
#include <sys/trchkid.h> /* for trace hook macros */

/* for timer support */
#include <sys/timers.h> /* for itimerspec, includes time.h */
#include <sys/timer.h> /* for TRBs */

/************************************************************
 TIMER CALLBACK ROUTINE
 This callback routine is invoked every time a timer expires.
 This executes in the context of decrementer interrupt handler.
************************************************************/
void timer_callback(struct trb *timer_req_blk_ptr)
{
TRCHKL2T(HKWD_USER1, 0xb, timer_req_blk_ptr->t_func_sdata);
}
```

The callback routine, `timer_callback`, has a TRB, which can contain some user supplied data, passed to it; in this case `t_func_sdata` is a signed integer containing, say, the device number. Any data that can't be passed as a parameter will have to be globally accessible; portions of a routine that access such shared resources are critical sections shared among threads and interrupt handlers and require appropriate synchronization.

The driver's top half, called `xyz_top.c`, differs from the sample driver in Chapter 1 in several ways described in the following discussion. One difference is the include files and declarations would include:

```
/* for timer support */
#include <sys/timers.h> /* for itimerspec, includes time.h */
#include <sys/timer.h> /* for TRBs */
#include <sys/m_intr.h> /* for INTTIMER priority */

extern void timer_callback();
struct trb *trbp; /* single pointer to TRB structure */
```

Note that the pointer to TRB is global and is therefore a shared resource requiring synchronization if the top half is to be reentrant. For simplicity, access to the pointer is not serialized in this example.

Here is what is added to the entry point `xyzopen`:

```
if((trbp = talloc()) == (struct trb *) NULL)
{ setuerror(ENOMEM); /* talloc failed, no space on pin heap */
 return(-1); /* setuerror() used: there is no ublock access */
}

trbp->id = thread_self(); /* or getpid() */
trbp->timerid = TIMERID_REAL;
trbp->eventlist = -1; /* no events being waited on */
trbp->func = (void (*)(void *))timer_callback;
trbp->t_func_sdata = (int) devno; /* or whatever data */
trbp->ipri = INTTIMER;
```

Note that `talloc` allocates space from the pinned heap. The fields of the TRB listed are the only ones a device driver would set; even so, most fields only need to be filled out if the callback routine requires them. `id`, `timerid`, `eventlist`, `t_func_sdata` are all for use by the callback routine. The callback routine, `timer_callback`, listed previously, currently makes no use of any of these fields.

Here is what is added to the entry point `xyzclose`:

```
tstop(trbp);
tfree(trbp);
```

Here is what is added to the entry point `xyzread`:

```
struct timespec threeSec;
struct itimerspec timeStruct;
...
threeSec.tv_sec = 3; /* 3 seconds */
threeSec.tv_nsec = 0; /* and no milliseconds */

timeStruct.it_interval = threeSec;
timeStruct.it_value = threeSec;

trbp->timeout = timeStruct;

tstart(trbp);
```

So, if a program issues a `read` on this device, it starts a timer that causes the callback routine to execute three seconds later.

Here is what is added to the CFG_INIT case section of `xyzconfig`:

```
if((return_code = pincode(timer_callback)) != 0)
{ setuerror(return_code);
return(-1);
}
```

This pins the bottom-half, containing the callback routine, in RAM.

Here is what is added to the CFG_TERM case section of `xyzconfig`:

```
if((return_code = unpincode(timer_callback)) != 0)
{ setuerror(return_code);
return(-1);
}
```

This enables the pages containing the bottom half to be subject to page replacement.

The makefile and `aprogram.c` are the same as in the watchdog timer example.

Now, when the kernel is extended, and the user program, `aprogram`, is run, a system trace shows that the callback routine executes three seconds after **read** is called:

```
010 trace 22.459560 UNDEFINED TRACE ID idx 0x21d4 traceid 0010
hookword 10E0000
type 0E
hookdata 0000 00000009 00630000 2FF97DC0 00000000 00000000

010 trace 25.459661 UNDEFINED TRACE ID idx 0x2210 traceid 0010
hookword 10E0000
type 0E
hookdata 0000 0000000B 00630000 00000000 00000000 00000000
```

Event Notification

An alternative to polling a device for when some data is available, or when some status is achieved, is to make use of any interrupts that the device generates and notify the driver that an interrupt was processed. In this case, the driver routine registers itself for an *event* and causes its thread to relinquish the system processor. Whenever the event is posted, the kernel awakens the driver's thread and the driver routine resumes execution.

The kernel service **et_wait** causes the calling thread to relinquish the system processor until that thread receives an event. The kernel service **et_post** enables a kernel process to post an event to some thread, so that the thread can resume execution. The *event* is a bit in a bit mask that is passed as a parameter to each call.

The following is an example of adding event notification to the real-time timer sample code given previously.

In `xyz_bot.c`, add to the declarations:

```
/* for event handling */
#define THE_EVENT (1 << 31) /* this is the high order bit */
```

There is an event mask that can be used to ensure that an event is not reserved for the base operating system kernel. Be aware that this event is specific to a 32-bit system.

Add the following line to the callback routine:

```
et_post(THE_EVENT, timer_req_blk_ptr->id);
```

The field `id`, in the TRB, contains the ID of the thread that had set the timer.

In `xyz_top.c` add to the declarations:

```
/* for event handling */
#include <sys/sleep.h> /* for EVENT_SIGRET flag */
#define THE_EVENT (1 << 31) /* this is a high order bit */
```

And add the following to the `xyzwrite` entry point:

```
et_wait(THE_EVENT, THE_EVENT, EVENT_SIGRET); /* clear event
                                               upon receipt */
TRCHKL1T(HKWD_USER1, 0x0a);
```

Keep `aprogram.c` the same as in "Sample Device Driver" on page 1-15. A portion of the trace looks like:

```
010 trace 18.261389 UNDEFINED TRACE ID idx 0x21c8 traceid 0010
hookword 10E0000
type 0E
hookdata 0000 0000000A 00630000 2FF97DC0 00000000 00000000

010 trace 21.261233 UNDEFINED TRACE ID idx 0x21fc traceid 0010
hookword 10E0000
type 0E
hookdata 0000 0000000B 00630000 00000000 00000000 00000000

010 trace 21.261919 UNDEFINED TRACE ID idx 0x2218 traceid 0010
hookword 10A0000
type 0A
hookdata 0000 0000000A
```

The trace shows entry into `xyzwrite`, the callback, and then the completion of `xyzwrite`.

Instead of posting events to a driver routine in a thread, a device driver may need to wait on an event and meanwhile make a shared resource available for another routine to lock. To do so, the driver can invoke the kernel service **e_sleep_thread**, which takes a pointer to an event and a pointer to a lock as parameters. If the event is to be posted from the interrupt side, be careful that the pointers refer to data in pinned memory.

An interrupt handler (or some other routine) may call the kernel service **e_wakeup**, to cause any threads sleeping on this event to resume execution and reacquire a lock on a shared resource.

If multiple locks are to be released and then reacquired, a driver will use the kernel services, **e_assert_wait**, **e_block_thread**, and **e_clear_wait** and **e_wakeup** instead.

For more information on event handling services, see *AIX Technical Reference, Volume 5: Kernel and Subsystems* and *AIX Technical Reference, Volume 6: Kernel and Subsystems*.

Serialization Services

Sharing data among multiple concurrent processes causes an inherent problem with maintaining data consistency. Consider, for example, a doubly linked list structure. In order to remove an element from this list, it is necessary to update pointers in both the preceding and succeeding the elements. During this update sequence the list is in an inconsistent state. If additional process activity to add or remove elements is allowed to proceed while in this state, the results would be unpredictable. Access could be serialized through the use of locks.

Device drivers have the following types of critical code sections:

- Critical code sections shared among process threads. (Interrupt disabling is not required.)
- Critical code sections shared among process threads and interrupt handlers. (Interrupt disabling is required.)

Uniprocessor (UP) Serialization

On a uniprocessor, concurrent access to shared data is achieved through preemption. A process can be preempted by another higher priority process, or by an interrupt routine.

Protection for access to data shared between process threads is provided by using locks. Protection for access to data shared between the base level and the interrupt level is provided by disabling interrupts.

Multiprocessing (MP) Serialization

Multiprocessing involves the use of more than a single processor. AIX implements shared memory symmetric multiprocessing, that is, all processors are functionally equivalent and can perform I/O and computations. AIX manages a pool of identical processors, any of which may be used to control I/O devices or reference any memory unit. Conflicts between processors attempting to access the same data at the same time are resolved by hardware instruction synchronization. Conflicts in access to shared memory structures are resolved by software synchronization techniques; most notably locking.

The multiprocessing discussion introduces the following new terminology:

Master Processor
: The default processor for of funneled code execution. This is usually the IPL boot processor.

Funneling
: Funneled code runs only on the master processor. This enables a UP driver to run unchanged by funneling its execution through one specific processor (the Master processor).

MP-safe
: Device driver code that can run on any processor. The code for the device driver provides for locks to serialize the device drivers execution. This provides for a coarse level of locking to enable parallel execution.

MP-efficient
: Device driver code that can run on any processor. The code for the device driver provides for locks to serialize access to devices and data structures. This provides for a finer level of locking to further enable parallel execution.

Masking interrupts or disabling the processor by calling the **i_disable** kernel service to serialize with an interrupt handler is no longer sufficient with symmetric multiprocessing (except for funneled code). The I/O is symmetric and interrupts can be routed to any of the processors in the complex; masking interrupts will not prevent the interrupt routine from executing simultaneously on another processor. Serialization in a symmetric multiprocessor system, therefore, requires the use of locks in addition to using **i_disable**.

Lock Overview

A number of serialization techniques are available defined by the intent to serialize access to critical code sections, or data items. Locking critical sections of code is a coarse locking level that supports MP-safe program execution. Locking at the data level is a finer locking level that supports either MP-safe or MP-efficient program execution. Additionally, atomic operations such as **fetch_and_add** and **compare_and_swap** can be used as an alternative to locks to provide reliable access to a single shared variable for reading or writing.

Lock contention wait can result in a process spinning on a busy lock or sleeping until the lock is granted.

AIX provides for the following types of locks:

Simple locks Spin locks that provide exclusive ownership and are not recursive. These locks are used for serialization among threads, serialization among threads and interrupt handlers, and serialization among threads and interrupt handlers.

Complex locks Sleep locks that provide read or write access and are recursive on request. These locks are used only for serialization among threads.

Lockl locks Sleeping, mutual exclusive locks provided for compatibility with AIX Version 3. These should not be used for newly written code.

Locking the global `kernel_lock` is not recommended (discouraged) because this lock has the potential to block the system scheduler. Also, this lock may be removed at some future time.

To support collecting and monitoring statistics about lock activity, AIX implements lock instrumentation. This function is activated or deactivated on an IPL boot basis via the **bosboot** command. Lock instrumentation criteria requires that locks be allocated before being used and deallocated when no longer needed. Instrumentation is not provided for **lockl** and **unlockl** lock services. Allocation calls and deallocation calls result in no operation if instrumentation is disabled.

Serializing Critical Sections

The following kernel services, which encapsulate interrupt control and simple locks, should be used to serialize critical sections in the driver bottom half:

disable_lock Raises the interrupt priority, and locks a simple lock if necessary.

unlock_enable Unlocks a simple lock if necessary, and restores the interrupt priority.

These kernel services should be used instead of calling the simple lock kernel services directly; this ensures that a thread will never be interrupted while it holds a simple lock. Allowing a thread which holds a simple lock to be interrupted can deadlock the system.

An MP-safe driver bottom half uses the **disable_lock** and **unlock_enable** kernel services to serialize device driver execution with a code lock. An MP-efficient driver bottom half replaces this code lock with one or more data locks, serializing access to individual data structures and devices instead of driver execution. Take care that this finer grained locking does not result in excessive execution path lengths.

Avoiding Lock Nesting

The **disable_lock** and **unlock_enable** kernel services cannot be nested on the same lock. An MP-safe or MP-efficient device driver must be structured to avoid the requirement for such nesting. One way to achieve this is to have two names for each routine which can be called from either outside or inside a critical section. The first name corresponds to a small routine which is called from outside a critical section; it acquires the lock, calls the second routine, and then releases the lock. The second routine assumes that the lock is already acquired.

Releasing Locks During Sleeps

A thread-interrupt lock must not be held across a sleep, or when calling an external routine which could sleep. Any kernel service which can be called from process level can result in a sleep, as can accessing data which is not pinned.

In the part of the driver bottom half which runs in the process environment with interrupts disabled, the **e_sleep_thread** kernel service can be used to sleep. This service releases a specified lock before sleeping, and reacquires the lock afterwards.

Ensuring Proper Lock Ordering

If a thread must acquire several locks, a strict lock ordering must be defined in order to avoid deadlocks. The locks generally must be acquired in the same order, and released in the reverse order. Typically, MP-safe device drivers have a single lock, so lock ordering is not normally a consideration for them. MP-efficient drivers must be carefully designed to prevent deadlocks due to lock ordering.

Device Driver Lock Models

The device driver lock model in an MP environment can take on one of the forms previously mentioned (funneled, MP-safe, or MP-efficient). The implementation model used by the device driver needs to be communicated to the kernel for the MP-safe or MP-efficient models. This is done by setting necessary flags in the appropriate data structures supporting the **devswadd**, **i_init**, and **iodone** kernel services. The funneled implementation is assumed by default.

The funneled model is essentially a UP implementation. This is appropriate for low throughput devices or migrating device drivers from the UP environment. This is the implementation model that is assumed if no changes are made to an existing UP device driver migrated to an MP environment. Serialization of critical code sections is accomplished by using the **lockl** and **unlockl** kernel services for sections shared among threads, and the **i_disable** and **i_enable** kernel services for sections shared among threads and interrupt handlers.

Although **lockl** locks are provided for compatibility with earlier releases, new code written for AIX Version 4.1 should use simple locks instead. This enables gathering lock instrumentation statistics for these locks.

The MP-safe model is intended for medium throughput devices and provides a coarse level of MP serialization without the complexity and overhead of MP-efficient implementation. Simple lock kernel services are provided for both serialization of critical sections shared among threads and serialization of critical sections shared among threads and interrupt handlers. This implementation enables critical sections of code to run on any processor in the complex, but not at the same time. The locking granularity is usually a single global lock at the device driver level.

The MP-efficient model is intended for high throughput devices by optimizing the parallel execution capabilities of an MP processor. A finer level of locking capability is available by using a hierarchy of locks based on data access serialization. Complex lock kernel services are provided for shared read and exclusive write access to data structures shared among

multiple-thread code sections. Simple lock kernel services are provided for serialization among thread and interrupt handler code sections.

The implementation model used depends on such factors as performance needs and whether the device driver is being migrated from a UP environment. If designing, or moving to an MP-safe or MP-efficient model, the recommendation is to start by using simple locks (a single global lock initially) and gradually refine the serialization as needed after identifying all the data structures shared between the top and bottom half of the driver. Also, analysis of the lock statistics provided by lock instrumentation can be used to identify contention bottlenecks that may point out the need for additional locking or the need for the read and write locking capability provided by complex locks.

MP-Safe Coding Sample

```
/* Skeleton code sample using the following lock related    */
/* kernel services:                                         */
/*                                                          */
/* lockl()            - conventional-lock lock request      */
/* lock_alloc()       - allocate simple lock                */
/* simple_lock_init() - initialize simple lock              */
/* lock_free()        - release simple lock                 */
/* unlockl()          - conventional-lock unlock request    */
/* devswadd()         - devsw table lock model options      */
/*                                                          */
/* (driver configuration lock definition and use)           */
...
lock_t config_lock ={LOCK_AVAIL};  /* define lockl lock     */
Simple_lock dd_lock;               /* define Simple MP lock */
...
int
dd_config(int cmd,
 struct uio *uiop)
{ /* start dd_config() */
 struct devsw dd_devsw;
 int rc;
 ...
 rc = lockl(&config_lock, LOCK_SHORT); /* dd_config lock */
 if (rc != LOCK_SUCC)
 return(EINVAL);
 lock_alloc(&dd_lock,        /* allocate Simple MP lock */
            LOCK_ALLOC_PIN,
            DD_LOCK,
             -1);
 simple_lock_init(&dd_lock); /* initialize Simple MP lock */
 ...
  switch(cmd)
 {
   case CFG_INIT:
   {
     dd_devsw.d_open = dd_open;
     dd_devsw.d_close = dd_close;
     dd_devsw.d_read = dd_read;
     dd_devsw.d_write = dd_write;
     dd_devsw.d_ioctl = dd_ioctl;
     dd_devsw.d_strategy = dd_strategy;
     dd_devsw.d_ttys = 0;
     dd_devsw.d_select = nodev;
     dd_devsw.d_config = dd_config;
     dd_devsw.d_print = nodev;
     dd_devsw.d_dump = nodev;
     dd_devsw.d_mpx = nodev;
     dd_devsw.d_revoke = nodev;
     dd_devsw.d_dsdptr = NULL;
     dd_devsw.d_selptr = NULL;
     dd_devsw.d_opts = DEV_MPSAFE;    /* register as MP SAFE */
     rc = devswadd(devno,&dd_devsw);  /* devsw table entry   */
     ...
    } /* end case CFG_INIT */
   break;
   case CFG_TERM:
   {
     ...
     lock_free(&dd_lock);            /* release MP lock        */
      ...
   } /* end case CFG_TERM */
   break;
```

```
  } /* end switch (cmd) */
 ...
 unlockl(&config_lock);          /* unlock dd_config lock     */
  return(SUCCESS);
} /* end dd_config() */
/* Skeleton code sample using the following lock related     */
/* kernel services:                                          */
/*                                                           */
/* simple_lock()   - set simple lock                         */
/* simple_unlock() - unlock simple lock                      */
/*                                                           */
/*  (thread -- thread process serialization)                 */
/*                                                           */
int
dd_read (dev_t devno, struct uio *uiop)
{ /* start dd_read() */
 simple_lock(&dd_lock);    /* set simple lock               */
 ...                       /* (critical section)            */
 simple_unlock(&dd_lock);  /* unlock simple lock            */
 return(SUCCESS);
} /* end dd_read */
/* Skeleton code sample using the following lock related     */
/* kernel services:                                          */
/*                                                           */
/* disable_lock()  - disable interrupts and set simple lock  */
/* unlock_enable() - unlock simple lock and enable interrupts*/
/*                                                           */
/* (thread -- interrupt process serialization)               */
/*                                                           */
int
dd_intr (void)
{ /* start dd_intr() */
 int old_level;        /* interrupt priority level save word */
 ...
 old_level = disable_lock(CURR_LEVEL, /* disable interrupts & */
                          &dd_lock);  /* set simple spin lock */
 ...                                  /* (critical section)   */
 unlock_enable(old_level,             /* unlock simple lock & */
               &dd_lock);             /* enable interrupts    */
 return(SUCCESS);
} /* end dd_intr */
```

MP-Efficient Coding Sample

```
/* Skeleton code sample using the following lock related     */
/* kernel services:                                          */
/*                                                           */
/* lockl()           - conventional-lock lock  request       */
/* lock_alloc()      - allocate Complex lock                 */
/* lock_init()       - initialize Complex lock               */
/* lock_free()       - release Complex lock                  */
/* unlockl()         - conventional-lock unlock request      */
 /*                                                           */
 /* (driver configuration lock definition and use)           */
...
lock_t config_lock = {LOCK_AVAIL};  /* dd_config lock        */
struct {               /* Adapater Control Block structure   */
   Complex_lock adapter_lock;       /* R/W Adapter lock      */
   struct Device_Ctl *next_device   /* device structure chain*/
} Adapter_Ctl;
struct {               /* Device Control Block structure     */
   Complex_lock device_lock;        /* W/Exclusive Device lock*/
    struct statistics device_stats; /* performance statistics */
   short dev_minor_no;             /* Device minor number    */
} Device_Ctl;
int
dd_config(int cmd,
          struct uio *uiop)
{ /* start dd_config() */
  /*
   * Use lockl operation to serialize the execution of the
    * config commands.
   */
  if ((rc = lockl(&config_lock, LOCK_SHORT)) != LOCK_SUCC) {
        return(EBUSY);
  }
  switch(cmd) {
      case CFG_INIT:
      {
      ...
      /*
          *  Define the locks in the adapter/device blocks
       */
         lock_alloc(&Adapter_Ctl.adapter_lock,
                    LOCK_ALLOC_PIN,
                    ADAPTER_CTL_LOCK,
                    -1);
         lock_init(&Adapter_Ctl.adapter_lock,
                   TRUE);
         lock_alloc(&Device_Ctl.device_lock,
                    LOCK_ALLOC_PIN,
                    DEVICE_CTL_LOCK,
                    Device_Ctl.dev_minor_no);
         lock_init(&Adapter_Ctl.adapter_lock,
                   TRUE);
         ...
      }  /* end case CFG_INIT */
      break;
      case CFG_TERM:
      {
       /*
        * Free the locks in the adapter/device control blocks
        */
         ...
         lock_free(&Adapter_Ctl.adapter_lock);
         lock_free(&Device_Ctl.device_lock);
```

```
        ...
      }  /* end case CFG_TERM */
      break;
      ...
  }
 unlockl(&config_lock);      /* unlock dd_config lock        */
 return (SUCCESS);
} /* end dd_config() */
/* Skeleton code sample using the following lock related     */
/* kernel services:                                          */
/*                                                           */
/* lock_read()  - lock Complex lock for shared read          */
/*                        - or -  write exclusive access     */
/* lock_write() - lock Complex lock for w/exclusive access   */
/* lock_done()  - release Complex lock                       */
/*                                                           */
/* (thread - thread process serialization only)              */
/*                                                           */
int
dd_ioctl(dev_t dev, int cmd, caddr_t arg, uint flag,
         chan_t chan, caddr_t  ext)
{
  int rc;                                /* return code        */
  ...
  /*
   * Lock adapter lock in shared read access (this is done to
   * protect against removal of device structure)
   */
    lock_read(&Adapter_Ctl.adapter_lock);
  ...
  switch(cmd) {
  /*
   * Return current device performance statistics
   */
        case DD_GET_STATS:
        {
         ...
        /*
         * Lock device control block for read access to gather
         * statistics for caller
         */
          lock_read(&Device_Ctl.device_lock);
         ...               /* (critical section)             */
          lock_done(&Device_Ctl.device_lock);
         rc = (SUCCESS);
         break;
        }
  /*
   * Update device performance statistics
   */
        case DD_UPDATE_STATS:
        {
         ...
        /*
         * Lock device control block for write access to update
         * statistics for caller
         */
         lock_write(&Device_Ctl.device_lock);
         ...               /* (critical section)             */
          lock_done(&Device_Ctl.device_lock);
         rc = (SUCCESS);
         break;
        }
  ...
```

```
    }
    lock_done(&Adapter_Ctl.adapter_lock);
    return(rc);
}
```

Making a Uniprocessor Device Driver Multiprocessor-Safe

The following list gives brief guidelines on porting an existing uniprocessor device driver to make it MP-safe or MP-efficient.

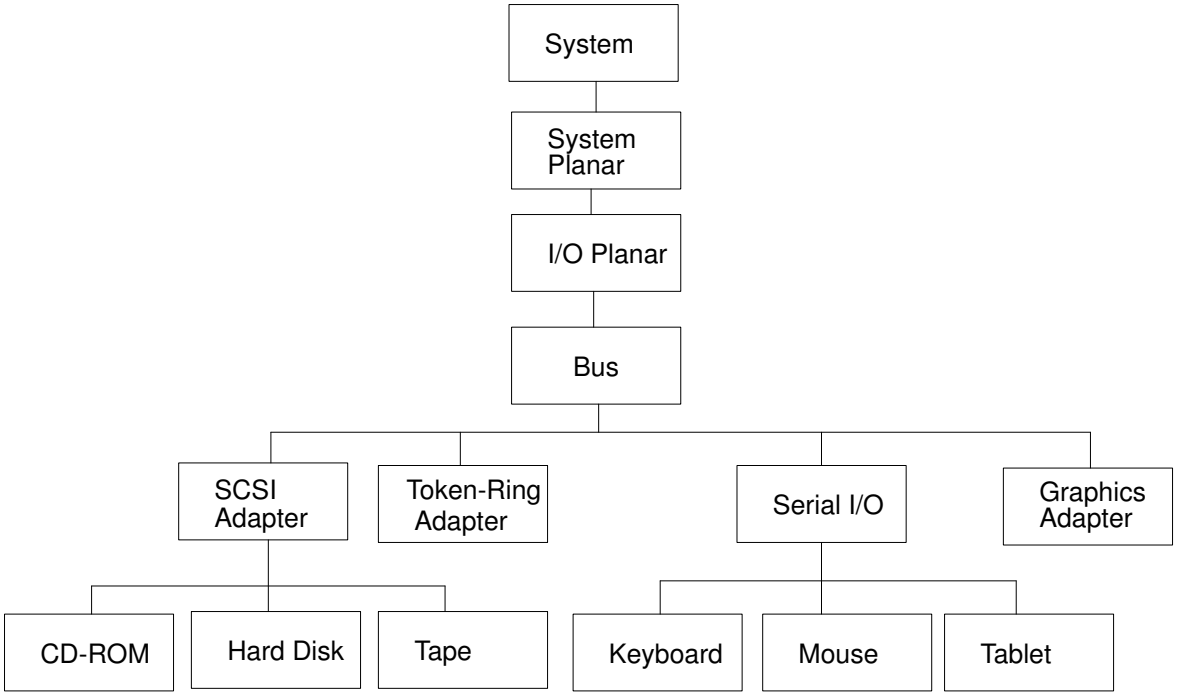
- Change the **devswadd**, **devstrat**, and **i_init** calls as necessary to indicate that the device driver strategy routine, iodone routine, and interrupt handlers are MP-safe or MP-efficient.
- If required, change the driver top half to use simple or complex locks instead of the **lockl** kernel service. If using simple locks, remove any nesting.
- In the driver bottom half:
  - Use **disable_lock** and **unlock_enable** instead of **i_disable** and **i_enable** for critical section serialization.
  - If the driver has nested calls to **i_disable**, remove the nesting, since the **disable_lock** service does not support nesting.
  - Ensure that locks are not held across sleeps. Modify the driver so that calls to the **e_sleep** service can be replaced with calls to the **e_sleep_thread** service. Release locks before using kernel services which are callable from process level. Do not access unpinned data while holding a lock.
- Ensure that locks are acquired and released in the correct order to prevent system deadlocks.
- If the driver relies on careful ordering of operations to avoid disabling interrupts (an example is a singly-linked list which is altered at process level and is always consistent so that it can be scanned at interrupt level), protect the operations with critical sections; such careful ordering does not provide sufficient protection on multiprocessor systems.
- It is possible that device driver routines which are called by the kernel with interrupts disabled do not call **i_disable** and **i_enable**. Ensure that these routines are serialized with calls to **disable_lock** and **unlock_enable**. The following device driver routines are called with interrupts disabled:
  - EPOW handlers (see “Early Power Off Warning” on page 3-9)
  - interrupt handlers (see Chapter 3, “Interrupts”)
  - timeout routines (see “Timer and Time of Day Services” in *AIX Kernel Extensions and Device Support Programming Concepts*)
  - watchdog routines (see “Timer and Time of Day Services” in *AIX Kernel Extensions and Device Support Programming Concepts*)
  - off-level handlers
  - iodone routines (see the **iodone** kernel service)
- Ensure that no thread-interrupt lock which could block the corresponding interrupt handler is held when the **i_init** and **i_clear** kernel services are called.
- Check the return values of the **tstop**, **w_init**, and **w_clear** timer and watchdog kernel services, and take appropriate action (such as releasing locks and retrying) if these services were unable to perform the requested operations.
- Ensure that if the driver has a **dddump** entry point, it makes no assumptions about the state of the device hardware when it is called.

Chapter 6. Device Configuration Methods

The dynamically loadable and unloadable aspect of the AIX Version 4.1 kernel requires that all device drivers have configuration methods to support the ability to load and unload them from the kernel. *Configuration methods* are sets of executables including a Define method, a Configure method, a Change method, an Unconfigure method, and an Undefine method. A *Configure method* is part of a set of *configuration methods*. Be careful to not confuse these terms. AIX Version 4.1 also has provisions for a Stop method and a Start method but these are seldom needed in the case of device drivers.

The System Device Hierarchy figure shows how relationships of devices to other devices on the system can be seen as a tree of parent-child relationships. Parents detect their children and then execute the appropriate configuration methods to introduce them to the system. These methods will then load the appropriate device drivers and make the device available for use.

The system's Configuration Manager (started by the **cfgmgr** command) actually oversees the entire configuration process by starting configuration methods, interpreting errors, and managing the configuration of child devices.



System Device Hierarchy

Device States

The system considers each device to be in one of several different *states* indicating the device's availability for use. The state of each device is stored in a database by the Object Data Manager (ODM). There are several databases used to maintain the configuration data for each device. The following is a list of states and their meanings to the system.

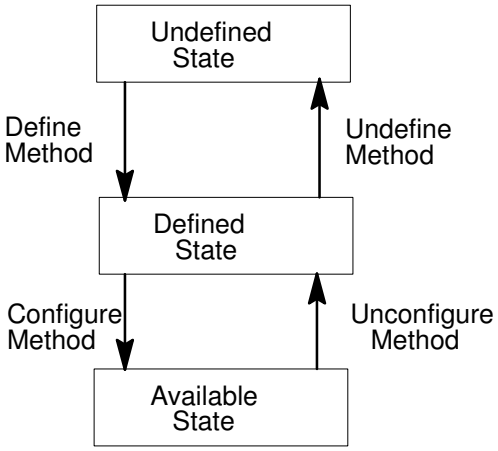
Undefined The device is not known to the system. There is no information about the device in the ODM database

Defined The device is known to the system but currently does not have its device driver loaded and is therefore not available for use

Available The device has its device driver loaded and is available for use

The various configuration methods will take a device from one state to another by detecting any attached devices, manipulating the databases, and loading or unloading the appropriate device driver. The names of a device's configuration methods are stored in the device's entry in the Predefined Device (PdDv) database.

The Device States and Methods figure shows how various methods change the state of a device.



Device States and Methods

ODM Configuration Databases

The databases used by the ODM (Object Data Manager) include the Predefined Devices (PdDv), Predefined Attributes (PdAt), Predefined Connections (PdCn), Customized Devices (CuDv), Customized Attributes (CuAt), Customized Dependencies (CuDep), Customized Device Drivers (CuDvDr), Customized Vital Product Data (CuVPD), and Config Rules (Config_Rules) object classes. The actual database files exist in the directories **/etc/objrepos** and **/usr/lib/objrepos**. The databases keep track of the devices defined to the system, the relationships among devices, and the various attribute values used to configure each device. The purpose of the various object classes is explained in the following list:

PdDv Contains definitions for the entire set of devices that could be supported by the system.

PdAt Contains attributes for the devices listed in PdDv.

PdCn Contains the supported connection points for parent-child devices.

CuDv Contains definitions for devices that have been introduced to the system and may contain devices that are in either the DEFINED or AVAILABLE state.

CuAt Contains the attributes required by the devices in CuDv.

CuDep Contains devices that are dependent on other devices.

CuDvDr Contains the major numbers of drivers loaded into the kernel as well as the device number (major, minor) of each device.

CuVPD Contains the Vital Product Data of a device if it contains such data.

Config_Rules Contains a list and order of config methods to be run.

Several commands and system calls exist to support manipulation of the various configuration databases. The commands include **odmget**, **odmadd**, **odmdelete**, **odmchange**, and **odmshow**.

The **odmshow** command is useful for displaying the format of the various databases.

The **odmget** command is useful for extracting current entries in each of the databases. These examples can then be used as templates for new entries that can be added with the odmadd command. When writing a new device driver package, new entries only need to be written for the PdDv, PdAt, PdCn, and possibly Config_Rules object classes.

The **odmadd** and **odmdelete** commands are useful for adding or deleting any new entries.

The system calls available for manipulating the configuration databases, usually called from within configuration methods, include **odm_initialize**, **odm_add_obj**, **odm_rm_obj**, **odm_change_obj**, **odm_get_first**, **odm_get_obj**, and **odm_terminate**. The various system calls will typically be used by the various configuration methods to view, add, change, and remove objects from the configuration databases.

Define Methods

Devices are first introduced to the system via a define method. A generic define method (/**usr**/**lib**/**methods**/**define**) on the system can be used for many devices. This method is designed to cover a wide variety of devices, but if a specific device requires special processing, a new specific define method will have to be written for it.

In general, a define method will take as parameters, the name of the new device, its *class*, *subclass*, and *type*, the name of its parent, and the new device's connection point. The define method will then verify these parameters and create a new entry in the CuDv object class for the device using the **odm_add_obj** subroutine. At this point, the device is only marked as Defined and the driver has not yet been loaded. A device's define method is typically invoked from the command line via the **mkdev** command. This command will actually look up the name of the define method to run from the PdDv database for the device being defined. The following is a description of the various flags for the generic define method:

–c *class*	Designates the class of the device.
–s *subclass*	Designates the subclass of the device.
–t *type*	Designates the type of the device.
–p *parent*	Designates the logical name of the device's parent.
–w *connection*	Designates the connection point.
–l *name*	Designates the new logical name of the device.
–u	Indicates that the **–l** flag is not allowed (the define method will generate a new logical name based on the device's prefix stored in PdDv).
–n	Indicates that the device has no parent or parent connection.
–o	Indicates that only one of these types of devices can exist and that the **–l** flag is not allowed.
–k	Indicates that the device can only be defined if one does not already exist at the specified connection point.

For a more detailed discussion on writing a define method, see "Device Configuration Subsystem" in *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts*.

Configure Methods

Config methods are designed to resolve a device's attributes, build device-dependent structures, find any child devices, and load a device's driver. A device may have attributes that could conflict with other devices that already exist on the system. For example, an adapter's bus-memory address cannot conflict or overlap the address of another adapter. It is the duty of the configure method to ensure that this does not occur. The **busresolve** routine (discussed in "Adapter Device Attributes and busresolve," on page 6-7), is a system call provided to ensure that bus attributes do not conflict.

A device dependent structure (DDS) is used by the configure method to pass vital information down to the device driver. This structure may contain information such as the bus slot location, DMA level, and interrupt level of an adapter, or various attributes needed to customize operation of the device. The DDS is built by reading in information from the various databases. For example, the slot number can be read from the `connwhere` value stored in CuDv. Various attributes required by the device driver can also be read from either PdAt or CuAt using the **getattr** subroutine. Because CuAt contains non-default attribute values, the **getattr** subroutine queries it first before retrieving values from PdAt.

The DDS structure is typically passed in to the driver through the driver's dd_config routine. After using the **loadext** routine to load the driver, the configure method should build the DDS structure and pass it to the driver using the **sysconfig** subroutine call and a command of CFG_INIT. The following is an example:

```
sysconfig (SYS_CFGDD, &dds, sizeof(dds));
```

The **sysconfig** routine will call the driver's **dd_config** routine with the CFG_INIT command and pass in the DDS structure.

For more information on the **sysconfig** routine, see *AIX Version 4.1 Technical Reference, Volume 2: Base Operating System and Extensions*. For more information on the **loadext** subroutine, see *AIX Version 4.1 Technical Reference, Volume 6: Kernel and Subsystems*.

If the device can have child devices connected to it, the configure method must detect them and run the child device's define method in order for the child to be added to the CuDv database. Once the define method has been successfully run, using the **odm_run_method** subroutine, the logical name of the newly created child should be printed to **stdout** so that the Configuration Manager can catch it and run the child's configure method.

Once the configure method has successfully loaded and configured the device driver, it should change the state of the device in the CuDv database to available using the **odm_change_obj** subroutine.

For a detailed discussion of configure methods, see "Device Configuration Subsystem" in *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts*.

Change Methods

Change methods are designed to modify a device's attributes to values other than the default. They can also alternately be designed to relocate a device to a different location or parent device. A generic change method /**usr**/**lib**/**methods**/**chgdevice** is provided on the system. A device's default attribute values are stored in the Predefined Attribute (PdAt) object class. The Customized Attribute (CuAt) object class is used to store values that are different from the default.

A change method should first verify that the attributes being modified are being changed to valid values. If the device is currently configured (available), it should be unconfigured by using the **odm_run_method** subroutine to execute the unconfigure method. This is required so that the device driver can later be initialized again with the new attribute values.

If an attribute is being changed to a value other than the default as listed in PdAt, a new attribute value should be added in CuAt. If a CuAt value already exists, it can be changed to the new value with the **odm_change_obj** subroutine.

If an attribute is being changed back to the default PdAt value, the CuAt entry should be deleted.

Once all the attributes have been changed, the device driver's configure method should be executed using the **odm_run_method** subroutine so that the device driver can be reloaded with the new attribute values. The following is a list of the various parameters for the generic change method:

–l *name* Designates the logical name of the device.

–p *parent* Designates the logical name of the device's new parent.

–w *connection* Designates the device's new connection point.

–a attr=*val*, attr=*val*
Designates the attribute names and new values.

For a more detailed discussion on writing a change method, see "Device Configuration Subsystem" in *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts*.

Unconfigure Methods

Unconfigure methods merely undo what a configure method has done. As in the case of the define and change methods, the generic unconfigure method (**/usr/lib/methods/ucfgdevice**) that has been provided should be satisfactory for many devices. However, if a specific device requires additional functionality, a new unconfigure method will have to be written.

In general, an unconfigure method should first verify that the device is in the correct state (AVAILABLE). It should then verify that any children of the device are also in the correct state (DEFINED). The unconfigure method should then call **sysconfig** with a command of CFG_TERM to instruct the device driver to terminate (the **sysconfig** routine will call the driver's **dd_config** routine with the CFG_TERM command). Once the device driver has successfully terminated, the driver can be removed with the **loadext** system call. As a final step, the unconfigure method should change the state of the device in CuDv from AVAILABLE to DEFINED. The following is a list of the various parameters for the generic unconfigure method:

–l <*name*> String that designates the logical name of the device

For more information on the **sysconfig** system call, please refer to the book *AIX Version 4.1 Technical Reference, Volume 2: Base Operating System and Extensions*. For reference information on the **loadext** subroutine, see *AIX Version 4.1 Technical Reference, Volume 6: Kernel and Subsystems*. For a more detailed discussion on writing an unconfigure method, see "Device Configuration Subsystem" in *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts*.

Undefine Methods

Undefine methods will take a device from the defined state to the undefined state by removing its entry from the CuDv database. In this state, all previous information about where the device was connected and what logical name was attached to it is removed. Any customized attributes for the device are also lost. As in the case of the change, define, and unconfigure methods, a generic define method (**/usr/lib/methods/undefine**), which should be satisfactory for many devices, has been provided.

In general, an undefine method should first verify that the device is in the correct state (DEFINED). It should then verify that child devices, if any, have been undefined (they don't exist in the CuDv database). The undefine method should then make sure that this device is not dependent on any other device, by checking the CuDep database, and removing all of its customized attributes from CuAt. The final step the undefine method should perform is to release the device number (devno) and remove any CuDvDr and CuDv entries. The following is a list of the various parameters for the generic undefine method:

–l <*name*> String that designates the logical name of the device

For a more detailed discussion on writing an undefine method, see "Device Configuration Subsystem" in *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts*.

Configuring Devices with No Parent

The configuration hierarchy is designed so that many devices are actually *children* of other *parent* devices. These child devices are automatically detected and defined by their parent configure methods. There are cases, however, of devices that are not connected to any parent devices. One example is the system node, **sys0**, which is a device that has no parent but is instead the parent and grandparent to most of the devices on the system. Many third-party vendor devices fall into this category of parent-less devices. In this case, these devices cannot rely on a parent because they are new to the system and the current OS either cannot detect them or will not recognize them.

When devices cannot rely on a parent to run their define method, they must use the Config_Rules database to initiate execution of their define method. Very often this define method will also have to detect the device being defined, which requires writing and using a new custom define method in place of the generic method. This is necessary because no parent exists or no parent has been written to perform the normal detection. The following is an example of a Config_Rules entry:

```
phase= 2
seq  = 50
boot_mask  = 0
rule = "/usr/lib/methods/defdevice"
```

In the previous example, the define method `/usr/lib/methods/defdevice`, would be run during phase 2 with a sequence number of 50. Phase 1 is used to configure basic essential devices needed to boot the system. Phase 2 is used to boot most base system devices. Note that a phase of 3 indicates methods to be run in phase 2 service mode. A high sequence number was chosen in case the device depended on other devices to be available. A high value causes the method to be run late in the configuration process. A nonzero boot mask, as defined in **/usr/include/sys/cfgdb.h**, indicates the type of boot (for example, disk, diskette, tape, or network) to which the method applies.

For more information on the Config_Rules Object Class, see *AIX Version 4.1 Technical Reference, Volume 6: Kernel and Subsystems*.

Adapter Device Attributes and busresolve

When configuring a device, care must be taken not to configure a device in a manner that conflicts with an existing device on the system. For example, a SCSI device cannot be configured with the same ID and LUN (logical unit number) as another device. In the case of adapters, not only must their connection locations (slot numbers) be unique, but several of their attributes must also be unique. For example, one adapter's allocated bus memory range must not overlap another's.

The **busresolve** routine is available to detect and resolve any possible conflicts with bus resource attributes. It is usually called from within an adapter's configure method. This routine needs to be called only if the configure method is being executed at run time. The bus configure method calls **busresolve** at boot time to properly resolve the attributes for all adapter devices.

busresolve will scan the CuAt and PdAt databases for all attributes with types that designate them as bus resource attributes. The following is a list of different bus attribute types:

Type	Description
O	Indicates an address and width for bus I/O.
M	Indicates an address and range for DMA transfers.
B	Indicates an address and range for non-DMA transfers.

A	Indicates a DMA arbitration level.
I	Indicates a sharable interrupt level.
N	Indicates a non-sharable interrupt level.
P	Indicates an interrupt priority class.
W	Specifies bus I/O and memory widths if not already specified by the address attribute.
G	Indicates a bus resource that must be assigned as part of a group.
S	Indicates a bus resource that must be shared with another adapter.

busresolve will adjust the attribute values within each attribute's specified allowable ranges. **busresolve** never adjusts the attributes of an AVAILABLE device. It only adjusts the attributes of DEFINED devices. This implies that devices configured first will have a greater probability of obtaining their default attribute values. When a configure method calls **busresolve** during run time, it should pass in the device's logical name as the *logname* parameter. This ensures that **busresolve** will adjust attributes only for the specified DEFINED device. Passing in NULL for *logname* causes **busresolve** to attempt resolution of attribute values for all Defined devices. This is typically done by the bus configuration manager at boot time and, therefore, does not need to be done by each individual adapter configure method. The CuAt and PdAt databases are automatically updated by **busresolve** once all attributes have been resolved.

Configuration of Devices on PCI and ISA Bus Systems

The configuration process for adapters attached to systems that contain PCI and ISA buses remains similar to the process for systems with a Micro Channel bus. The same ODM and configuration methods scheme is used with one additional step required from the system administrator when configuring adapters attached to the ISA bus. The PCI and ISA bus configuration also differs from the Micro Channel bus in that the ISA is actually treated as a child of the PCI bus even though the ISA bus has adapter children of its own. During configuration of the PCI bus, the ISA bus is detected and then a separate configure method is called for the ISA bus.

Because ISA adapters do not implement the notion of POS registers as in the case of Micro Channel adapters, an additional step must be performed by the system administrator. In the case of PCI and Micro Channel adapters, through the machine device driver, the bus configure method can detect the type of adapter attached and then dynamically set certain parameters such as the interrupt level or DMA arbitration level. In the case of ISA adapters, these types of parameters are set via jumpers or DIP switches that physically reside on the adapter. The system administrator installing the card must first set these parameters to avoid conflicts with other adapters before physically installing the adapter into the system. Since these settings cannot be detected on the ISA bus, the system administrator must manually add the CuDv (Customized Device) and CuAt (Customized Attribute) entries for the adapter. The CuDv entries should reflect that the adapter is a child of the ISA bus and its location code must correspond to the correct slot in which it resides. The CuAt entries must also reflect the actual settings on the adapter itself and must also not conflict with other adapters on the ISA bus. The **generic** field of the CuAt and PdAt entries should also contain a second character **U** to indicate that the particular attribute value is user modified. Newer ISA adapters that provide dynamic detection and modification of card settings are currently not supported. ISA adapters that require dynamic modification of settings on the card will require the device driver to perform this function, rather than the configuration methods.

Once the ISA specific requirements have been met, the configuration process of the ISA and PCI busses is identical to that of the Micro Channel bus as previously discussed. The **busresolve** routine is called by the corresponding bus configure method during boot so that adapter configure methods are not required to call it. The only time an adapter configure

method should call **busresolve** is during a runtime configure when the current attributes of the adapter being configured must be validated against values already resolved during boot.

Configuration of Devices on PCMCIA Systems

Although configuration of adapters attached to a PCMCIA bus is, in general, similar to that of MCA systems, there are specific differences that must be addressed. The main difference lies in the interdependency of an adapter's configuration methods, device driver **ddconfig** routine, and the card service kernel extension. These pieces are all necessary to successfully integrate the AIX Version 4.1 configuration subsystem with the PCMCIA interface.

The PCMCIA bus is viewed as a child of the ISA bus. Therefore, when the ISA bus is configured, via **cfgbus_isa**, the PCMCIA bus is detected and its configure method, **cfgbus_pcmcia**, is executed. The logical names of the ISA and PCMCIA busses are bus1 and bus2, respectively. The PCMCIA bus will have a location code of 00–20 (for bus2). Various steps are completed during configuration of the PCMCIA bus. These include:

- Adding the correct bus_id attribute into the PdAt database
- Loading the card and socket services extensions into the kernel
- Defining children of the PCMCIA bus and printing their logical names to **stdout** so that they can be configured

Each card on the PCMCIA bus will have a location code of 00–*XY* where *X* is usually 2 (for bus2) and *Y* is the slot number.

Insertion and removal of PCMCIA cards are automatically detected. The **acfgd** command automatically configures or unconfigures PCMCIA devices. The **acfgd** command can execute additional scripts before and after it executes the configuration or unconfiguration method. Although device drivers are aware of card insertion or removal, applications are not aware of them. If an application does not handle error cases on I/O system calls, the application can hang after card removal.

One additional attribute is also required for PCMCIA devices. To aid in detection of devices plugged into the PCMCIA bus, all device PdAt entries should have an attribute named `"pcmcia_devid"`. This attribute is similar to the **devid** field in the PdDv entry but differs in that this attribute's value specifies which tuple of the card to use and what value or values are allowed. The following is an example PdAt stanza:

```
uniquetype      = <class/subclass/type>
attribute       = "pcmcia_devid"
deflt           = "ttoo,value1:ttoo,value1,value2,value3"
values          = ""
width           = ""
type            = "R"
generic         = "D"
rep             = "s"
nls_index       = ""
```

The `deflt` string (limited to 255 characters) contains the values that are to matched with the tuple values read from the PCMCIA device. The following describes the syntax for the `deflt` field:

`tt` Tuple code (in hexadecimal) that identifies the tuple to be used for the comparison

`oo` Offset (in hexadecimal) into the tuple from which to begin the comparison

`value1,value2,...` The value(s) that are to be matched

Note: The example contains a `:` (colon), which indicates that there are 2 sets to be matched. Both sets must match for a positive identification. If only 1 set is required, the `:` (colon) and second set of values would not be needed. Since most PCMCIA

devices contain a Manufacturing ID tuple, this is the recommended tuple to use for the match.

The card services kernel extension can be accessed by device drivers through the **CardServices** kernel service. An extended system call, **svcCardServices**, is provided for use by configuration methods. The **CardServices** kernel service is used to perform several functions on a PCMCIA device such as detecting the device, reading its tuples, and registering the device. The following is a list of supported card service functions:

- AccessConfigurationRegister
- DeregisterClient
- GetCardServicesInfo
- GetClientInfo
- GetConfigurationInfo
- GetEventMask
- GetFirstClient
- GetNextClient
- GetFirstTuple
- GetNextTuple
- GetStatus
- GetTupleData
- MapLogSocket
- MapPhySocket
- MapLogWindow
- MapPhyWindow
- MapMemPage
- ModifyConfiguration
- ModifyWindow
- RegisterClient
- ReleaseConfiguration
- ReleaseExclusive
- RequestExclusive
- ReleaseIO
- RequestIO
- ReleaseIRQ
- ReleaseSocketMask
- ReleaseWindow
- RequestConfiguration
- RequestIRQ
- RequestSocketMask
- RequestWindow
- ResetCard
- SetEventMask
- ValidateCIS
- VendorSpecific

The following is a list of functions that are defined in the PCMCIA Card Services Interface Specification Release 2.10, but are not supported:

- OpenMemory
- ReadMemory
- WriteMemory
- CopyMemory
- RegisterEraseQueue
- CheckEraseQueue
- DeregisterEraseQueue
- CloseMemory
- GetFirstRegion
- GetNextRegion
- GetFirstPartition
- GetNextPartition

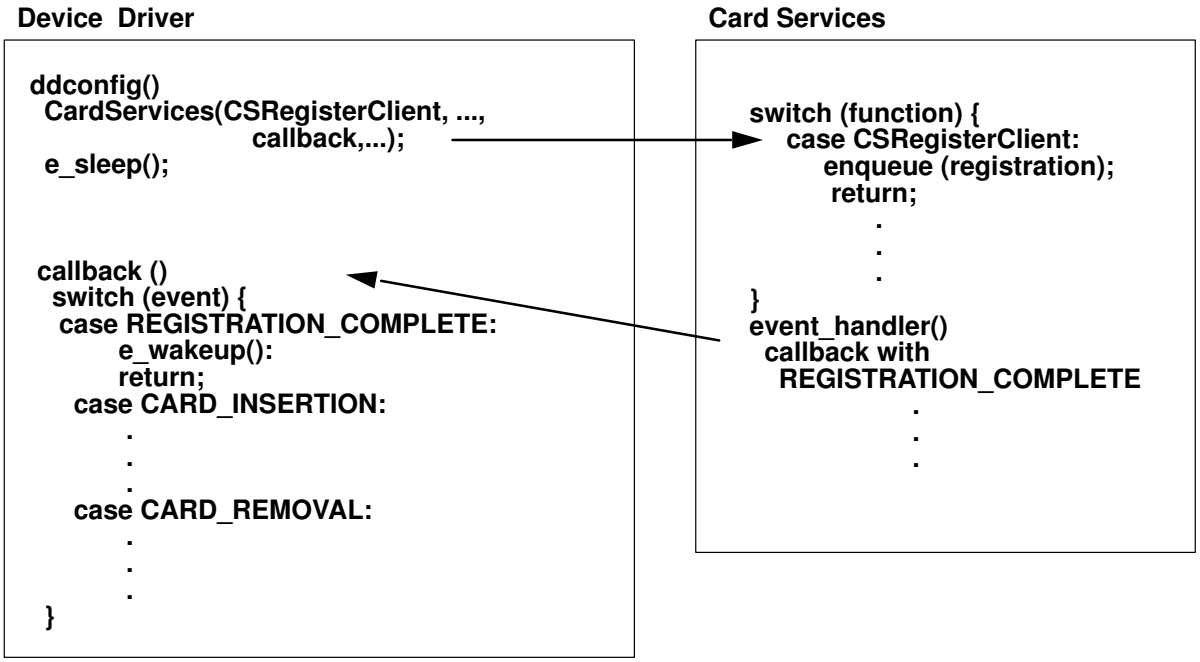
- ReturnSSEntry
- RegisterMTD
- RegisterTimer
- SetRegion
- ReplaceSocketServices
- AdjustResourceInfo

To access a PCMCIA socket through card services, a device driver should know the logical socket number. The configure method for a PCMCIA device should pass it to a device driver by the MapPhySocket function through **svcCardServices.** MapPhySocket needs the physical socket number and the physical adapter number. The physical socket number is `connection code` –1. The physical adapter number is the device number for the PCMCIA bus (bus2).

Card service function codes are prefixed by “CS,” for example, CSRegisterClient. Event codes are prefixed by “CSE_,” for example, CSE_CARD_INSERTION. Error codes are prefixed by “CSR_,” for example, CSR_SUCCESS. These function codes are defined in **sys.pcmciacs.h**.

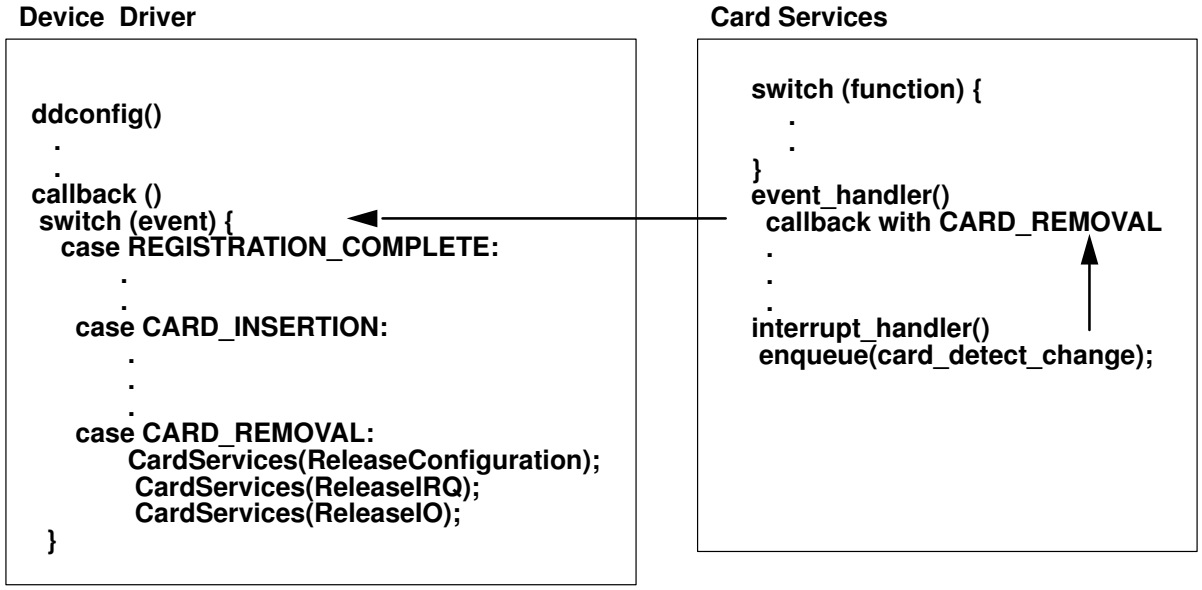
PCMCIA device drivers also require a callback routine whose address is passed to the **CardServices** routine when needed. This callback routine is called by the card services kernel extension as notification to the driver when certain events occur. Each PCMCIA device driver must process REGISTRATION_COMPLETE, CARD_INSERTION, and CARD_REMOVAL events. For further explanation on these events, see the PCMCIA Card Services Interface Specification Release 2.10.

During initialization of the driver in the **ddconfig** routine, the CSRegisterClient function should be called to register the device. The **ddconfig** routine should then sleep. When the card services extension has finished registration of the device, it will call the callback routine with a REGISTRATION_COMPLETE event notification. The driver can then be awakened. The following figure shows the interaction between the **ddconfig** routine and card services initialization. In a normal case, the CARD_INSERTION event comes before the REGISTRATION_COMPLETE event, and a driver calls the RequestXXX functions in the callback routine. After the REGISTRATION_COMPLETE event is called and the configure method’s thread is awakened, the driver can initialize hardware in the PCMCIA card, if the CARD_INSERTION event and the RequestXXX function were called successfully.



Device Driver and Card Services Interaction during Initialization

When a card is inserted or removed from a socket, an interrupt is generated that causes the card services extension to call the callback routine with a CARD_INSERTION or CARD_REMOVAL event. The driver then takes the appropriate action. Most drivers call the RequestXXX functions on CARD_INSERTION and the ReleaseXXX functions on CARD_REMOVAL. After these events are processed by all PCMCIA device drivers, the **acfgd** command configures or unconfigures devices. The following figure shows the interaction between the Card Services kernel extension and the device driver when a PCMCIA card is removed.



Device Driver and Card Services Interaction during Card Removal

Processing Pending or Wrong Interrupts after PCMCIA Card Removal

Wrong I/O interrupts can happen when a card is removed. It can be sent to a device driver faster than the CARD_REMOVAL event, because CARD_REMOVAL is sent by a kernel process. A device driver should check to see if the card really exists at the top of the I/O interrupt routine. Also, an interrupt can be pending before a card is removed. A device driver must handle these situations. A data storage exception is likely to happen because the pointer address indexed with the I/O register values cannot be translated to a real address.

Critical Section on Configuring/Unconfiguring a PCMCIA Card

In AIX, multiple threads can call card services. When configuring or unconfiguring a PCMCIA card, system hangs can occur if the other threads try to access the configuring/unconfiguring card.

A thread that configures or unconfigures a PCMCIA card should prevent other threads from accessing the card by holding the lock in card services.

After the critical section ends, the thread that has the lock must release the lock. It prevents the event handler in card services, auto–config daemon, configure methods for PCMCIA devices, and PCMCIA GUI from working normally. One of the VendorSpecific functions, CSaixLockSocket, is prepared. This function is defined in **sys/pcmciacsAix.h**.

Creating and Releasing Major and Minor Numbers for a Special File

Devices are generally identified in the kernel through major and minor numbers. Usually, a major number identifies a particular device driver. Minor numbers identify specific device instances known to the device driver. However, a device driver can be assigned multiple major numbers. Also, minor numbers can be used to identify different modes of operation for a specific 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 by opening the corresponding device special file located in the /**dev** directory. The special file i-node 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 many subdevices supported by the device driver. The minor device number usually serves as an index into a device driver-maintained array of information about each of many devices or subdevices supported by the device driver.

Creating 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 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** device configuration subroutine is called with a particular device driver instance name passed as a parameter, it performs the following actions:

- 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 just call the **genmajor** subroutine with the device's device driver instance name. If the device has not been assigned a major number, the **genmajor** subroutine 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 CuDv object. For most devices, the device driver instance name is simply the device driver name. If the device drive uses multiple major numbers, a different device driver instance name must be assigned for each major number.

Creating Minor Numbers

The allocation of device minor numbers is highly device-specific. A device Configure method can determine minor number assignments on its own or use the **genminor** and **getminor** device configuration subroutines. When the **genminor** subroutine is used to allocate minor numbers for a device, information is stored in the configuration database. This database keeps track of what minor numbers have been assigned for a particular major number, and the minor numbers being assigned to the device. The **getminor** subroutine 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, thus eliminating 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 special files. If the device minor numbers were allocated with the **genminor** subroutine, the **reldevno** device configuration subroutine can be used to both delete the major and minor number assignments and to delete the special files.

Chapter 7. Block Device Drivers

A block device driver interacts with a special facility in the kernel called the *buffer cache*. Special entry points in the driver are provided because of this interaction. A block device driver may also support character type interaction through read and write operations referred to as *raw* I/O. The principal characteristic of block devices is to perform I/O operations using system facilities such as buffer cache management and paging.

Data read from character devices is *not* stored in a cache for subsequent reading from system buffers. For block device drivers, data *is* stored in a cache. Block devices interact with the system to keep the cache containing information that a process (or multiple processes) can read from at any time. If the information is not in the cache, the system (not the user) requests the data from the block device driver.

Like all devices, the interaction with block devices is through shared memory. In addition, there are routines to indicate when data in the shared memory (called **buf** structures) has completed I/O processing.

The following sections cover the entry points responsible for the movement of data to and from block devices. This includes control information, the shared memory facilities, the mechanisms for programs to share the information, and the use of the kernel cache.

Finally, it may be necessary to talk to the device directly without interacting with the system buffer cache. This topic is presented in “Character Access to Block Device Drivers” on page 7-6.

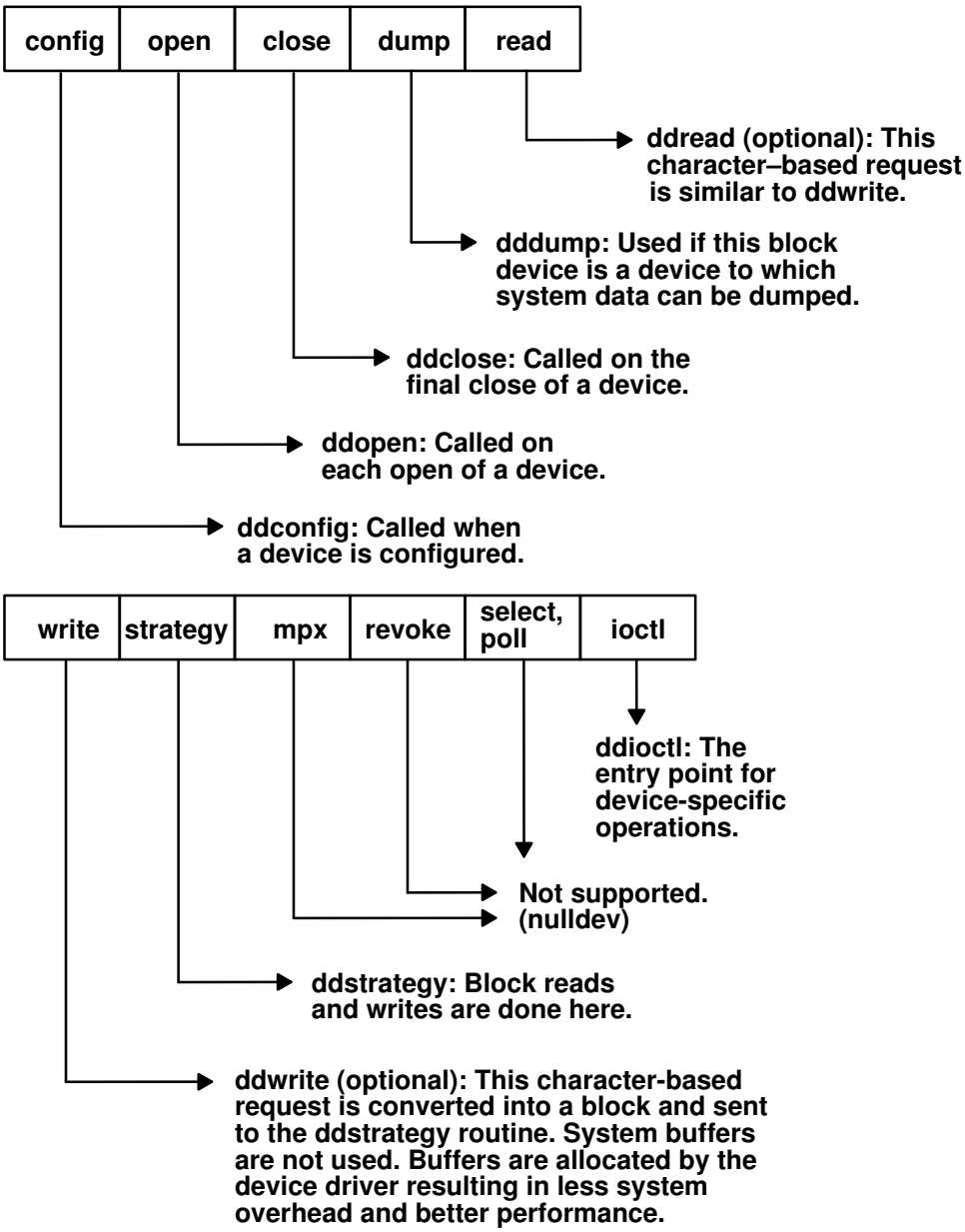
Block I/O Device Driver Entry Points

The device switch table contains the entry point addresses of the interface routines for each device driver in the system, just as it does for the character device drivers. The following Entry Points for the Block Device Driver figure shows the entry points for a block device driver.

Like the character device driver, the block device driver must supply a **config** routine for configuration support as well as an **open** and a **close** routine. The **open** routine is called each time the device is opened and the **close** routine is called only on the final close of the device.

Instead of having separate **read** and **write** routines, like character device drivers, each block device driver has a **strategy** routine. This routine is called with a pointer to a buffer header, known as the **buf** structure, which contains the I/O request parameter.

The **strategy** routine handles requests as buffers to be written or read from the device.



Entry Points for a Block Device Driver

ddconfig Entry Point

The configuration routine of a block device driver creates an entry in the device switch table for the block device driver. The device may support raw access. Raw access is character access to a block device. In this case, the driver's configure method must have created **/dev** special file entries for use in raw access. These special files retain the same major and minor numbers as their corresponding block device special files, but they have the letter *r* as a prefix, and the special files are created as character rather than as block.

For example, a block device named **/dev/hdisk0** that supports raw access also has a **/dev/rhdisk0** special file. The system calls the **read** and **write** routines of the raw device if **/dev/rhdisk0** is opened. Other UNIX systems may not allocate the same major and minor numbers for both character and block devices.

ddopen and ddclose Entry Points

The AIX operating system supports only a few block devices in normal installation. These devices, such as hard disks and CD-ROM, are capable of random access and are opened by system services such as the buffer cache and paging subsystem. They are not to be opened directly by user space applications during normal system operations, but may be opened during maintenance by applications such as **fsck**.

The **ddopen** routine verifies that the device is a valid device and that it is online and available. It also performs any setup required by the driver, such as pinning of code and allocation of data structures.

Most of the block devices are attached to the SCSI adapter, and open the SCSI adapter device driver to communicate with the device. For more information on the SCSI subsystem, see the chapter on small computer system interface (SCSI) subsystem in *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts* and the chapter on SCSI device drivers in this book.

The device performs **ddclose** processing to release the resource. If the device is attached to the SCSI bus, refer to the *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts* book and the chapter on SCSI device drivers in this book for details.

ddstrategy Entry Point

The I/O requests to the physical device are accomplished through the **strategy** routine. The **strategy** routine provides a “strategy” for mapping I/O requests to the device so that it minimizes requests to the device and maximizes data transfer. When the **strategy** routine (**ddstrategy** device driver entry point) is called, a pointer to a buffer header or a chain of buffer headers specifies the request for device I/O. The **strategy** entry point is called in a user process context when the buffer cache does not contain the buffer requested by the user. The **strategy** routine, however, does not know about the user process.

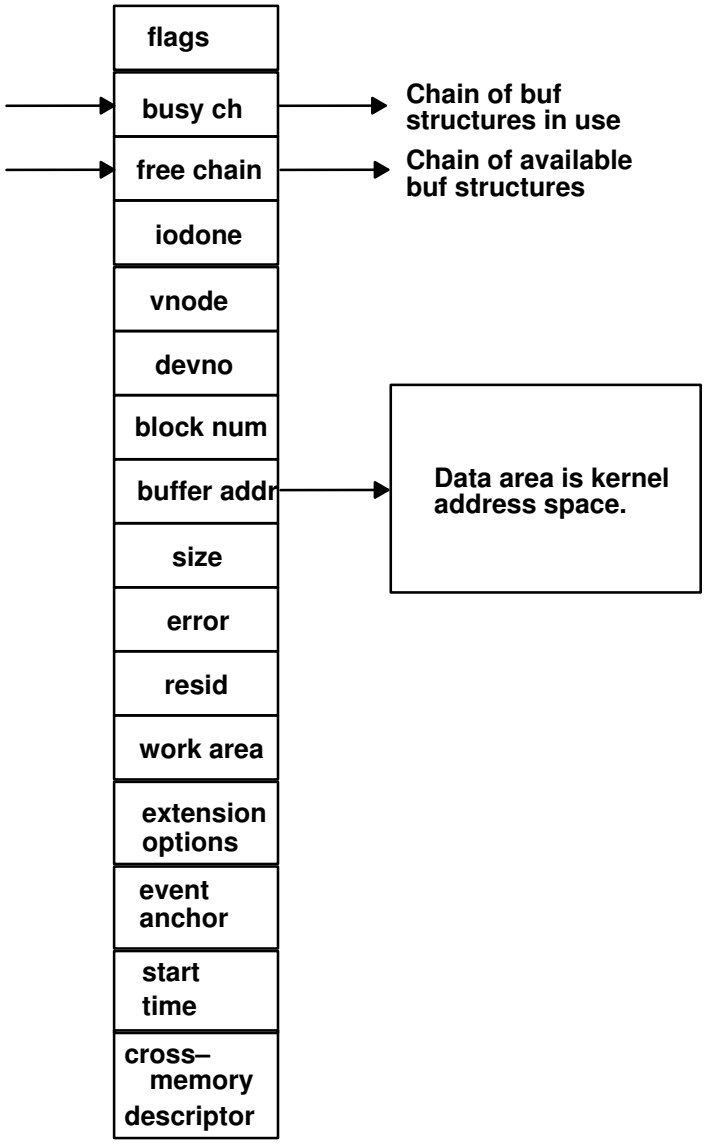
The buffer header contains the following information:

- The major and the minor numbers of the device
- The description of the memory buffer to be used for the data transfer
- The direction of the transfer
- The transfer count
- The block number on the device for which the transfer is targeted
- The operation flag

The **strategy** routine returns to the caller as soon as the buffer headers are queued to the appropriate device queue. Note that the **strategy** routine provides no return code to the caller and never waits for I/O completion before returning. This means that all requests are assumed valid in terms of parameters and that the request is asynchronous. Normal errors, such as out-of-range blocks, are caught but not returned directly as a return code.

The execution of the request completes some time later. The buffer structure contains fields for reporting the completion of the request.

A header contains all the information required to perform block I/O. The **buf** structure is shown in the buf Structure figure. It is the primary interface to the bottom half of block device drivers.



buf Structure

In AIX, the traditional **strategy** interface is extended as follows:

- The device driver **strategy** routine is called with a list of **buf** structures, chained using the *av_forw* and *av_back* pointers. The last entry in this list has a NULL *av_forw* pointer.
- When the operation is completed and the driver calls the **iodone** kernel service, the **b_iodone** function defined by the caller is scheduled to run as a software interrupt handler.

The **buf** structure and its associated data page must be pinned before calling the **strategy** routine. This is outlined in the **/usr/include/sys/buf.h** include file.

The **buf** structure contains the operation to be performed and status information to be returned to the caller, and is more like a message exchanged between a requestor and a service provider.

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 services. A residual count of the number of bytes requested but not transferred by the operation is placed in the `b_resid` field of the **buf** structure by the device driver before the I/O is marked as complete for the buffer header. If all the requested bytes are transferred, this count is set to zero.

For more information on the specific fields of the **buf** structure, see “buf Structure” in *AIX Version 4.1 Technical Reference, Volume 5: Kernel and Subsystems*.

Note: In AIX Version 4.1 a new flag, B_MPSAFE, has been added to the list of valid flags for the b_flags field of the **buf** structure definition. This flag is to indicate whether or not it is safe for the **iodone** processing to be performed on one processor or multiple processors. The **devstrat** kernel service will actually mark the **buf** structure with the B_MPSAFE_INITIAL flag once the B_MPSAFE flag has been sent. A device driver’s strategy routine, however, does not have to be concerned with this flag, since it is merely an instruction to the **iodone** system call about how to complete the processing of the **buf** structure.

Reordering Block I/O Requests

Multiple I/O requests can also be presented to the **strategy** routine, where the additional buffer headers can be chained to the first by using the *av_forw* pointers. While the device **strategy** routine is free to rearrange the buffers on its device queue with respect to the processing of single request, the ordering of the buffer headers provided in a chain to the **strategy** routine cannot be modified. The **strategy** routine also determines if the block number requested is valid for the device. In the case of a read-only operation, a block number at the end-of-media is not considered as an error, but no data is transferred.

For a write operation, if the block number is at the end-of-media, it is considered an error, the **B_ERROR** flag in the **buf** structure is set, and the `b_error` field contains the **ENXIO** value.

Categorizing Requests 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 the system memory associated with the device. The data structure contains the device status along with the device error information and the device queue pointers. Some device drivers maintain more than one queue of buffer headers. For example, one queue for the requests that are waiting for I/O start and another queue for the request that are currently in process.

For SCSI operations, the queueing process scans the pending queue for the requested device so that the number of SCSI operations is minimized. The requests are grouped by one of the following rules:

- Contiguous write operations
- Operations larger than maximum transfer size
- Operations requiring special processing

The coalesced (grouped) requests will be removed from the pending queue and placed in the **in_progress** queue so that a single command can be built to satisfy the requests. These requests are then queued and the routine to start I/O is called.

Starting Processing with the Start I/O Routine

The routine to start I/O checks to make sure that the device is not busy, and then scans the request queues in an attempt to find an operation to start.

First, the command stack is checked to see if a command needs to be restarted. Then the **in_progress** queue is checked to start any operations that have already been coalesced. Finally, the **pending** queue is checked. If it is not empty, the **coalesce** routine is called to group the operations into the **in_progress** queue.

When a request has been found and built, the adapter device driver is called by the strategy routine to begin processing the operation. While the queues are being scanned and an operation is in progress, the device busy flag is set. It is then reset if no request is found.

Once the I/O handling routine has completed an I/O transfer, it calls the **iodone** kernel service that determines if the indicated operation has completed successfully or if it has failed. If the operation is successful and complete, the next request is processed by the start

I/O routine. If the operation has failed, your general failure processing routine is called in an attempt (such as retry) to clear the error.

dddump Entry Point

To support system dumps, a block device driver must supply the **dddump** entry point. It is called by the **devdump** kernel service. See “Debugging Tools” on page 15-1 for more information on system dumps. See “SCSI Device Drivers” on page 8-1 for more information on providing system dump support on SCSI devices.

Character Access to Block Device Drivers

While a character device driver can only be accessed by a character special file, most block device drivers provide both a block and a character special file. With this dual interface, a user can access the device in either block or character mode.

Note that the block device driver must have a **read** and a **write** entry point as well as a **strategy** entry point if it supports both character and block mode access. If it supports only block mode, it only needs to support a **strategy** entry point.

The diskette or hard disk device drivers are examples of the dual nature of block device drivers. The diskette is accessed by /**dev**/**fd0** for block mode and by /**dev**/**rfd0** for character (raw) mode. The hard disk is accessed by /**dev**/**hdisk0** for block mode and /**dev**/**rhdisk0** for character (raw) mode.

Raw I/O Processing

Raw I/O processing is a mechanism by which a block device driver has the ability to transfer data without using the I/O buffer cache. The raw I/O request is converted into a block and then sent to the device driver **strategy** entry point to be processed while the **read** and the **write** routines are typically waiting for the I/O completion.

When your device driver is configured, it contains entries for both **read** and **write** (raw access) and **strategy** (block access) routines. In addition, the configure method must set up the /**dev** entries for both special files.

If there is no buffer cache and you make the request directly, a different buffering facility is involved. You are providing a buffer passed in through the uio services. Therefore the **read** and **write** entry points are talking to a user process and translating the requests into **strategy** requests but still using **buf** structures. Because the **buf** structure contains a header that contains a pointer to the data area, it can be mapped to point to a user data area.

In fact, the user buffer can come out of the user data, text segments, shared memory segments, or the system segment. The different areas are defined in the **uio** and **iovec** structures.

The **read** and **write** routines of the raw device driver use the **uphysio** kernel service to map the **uio** areas into **buf** structures used by the strategy routines. After filling in the **buf** structure with data passed to it through the **uio** structure, the **uphysio** kernel service will call the block device driver’s strategy routine. The number of **buf** headers sent is determined by the `buf_cnt` parameter passed to the **uphysio** kernel service. The **uphysio** kernel service returns only after all I/O has completed or after encountering an error. See “uphysio Kernel Service” in *AIX Version 4.1 Technical Reference, Volume 5: Kernel and Subsystems* for a more detailed discussion of this kernel service.

Note: Use care when accessing a block device through its character interface. Because the buffer cache is bypassed, the driver must be sure that no data currently exists in the kernel buffer cache. If another process did have data present in the cache, there is a high probability of data becoming inconsistent with data obtained through the character interface.

Block I/O Device Summary

A block I/O device contains a device name for its block device and its optional character device. Block devices support **strategy** routines, and possibly support **read** and **write** routines. The kernel cache speeds up access to data by allowing multiple processes to use the same data and keeping data that is referenced often in the cache. However, the cache is based on buffer sizes compatible with UNIX file system block sizes and is not efficient for applications that can use larger block sizes. To improve performance, a block device driver also provides raw or character access to block devices. More information is available in *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts.*

Chapter 8. SCSI Device Drivers

The AIX Small Computer Systems Interface (SCSI) subsystem has two parts:

- SCSI Device Driver
- SCSI Adapter Device Driver

The SCSI adapter device driver is designed to shield you from having to communicate directly with the system I/O hardware. This gives you the ability to successfully write a SCSI device driver without having a detailed knowledge of the system hardware. You can look at the SCSI subsystem as a two-tiered structure in which the adapter device driver is the bottom or supporting layer. As a programmer, you need only worry about the upper layer. This chapter only discusses writing a SCSI device driver, because the SCSI adapter device driver is already provided in AIX.

The SCSI adapter device driver, or lower layer, is responsible only for the communications to and from the SCSI bus, and any error logging and recovery. The upper layer is responsible for all of the device-specific commands. The SCSI device driver should handle all commands directed towards its specific device by building the necessary sequence of I/O requests to the SCSI adapter device driver in order to properly communicate with the device.

These I/O requests contain the SCSI commands that are needed by the SCSI device. One important aspect to note is that the SCSI device driver cannot access any of the adapter resources and should never try to pass the SCSI commands directly to the adapter, since it has absolutely no knowledge of the meaning of those commands.

SCSI Device Driver Overview

The role of the SCSI device driver is to pass information between the operating system and the SCSI adapter device driver by accepting I/O requests and passing these requests to the SCSI adapter device driver. The device driver should accept either character or block I/O requests, build the necessary SCSI commands, and then issue these commands to the device through the SCSI adapter device driver.

The SCSI device driver should also process the various required reservations and releases needed for the device. The device driver is notified through the **iodone** kernel service once the adapter has completed the processing of the command. The device driver should then notify its calling process that the request has completed processing through the **iodone** kernel service.

SCSI Adapter Device Driver Overview

Unlike most other device drivers, the SCSI adapter device driver does *not* support the **read** and **write** subroutines. It only supports the **open**, **close**, **ioctl**, **config**, and **strategy** subroutines. Included with the **open** subroutine call is the **openx** subroutine that allows SCSI adapter diagnostics.

A SCSI device driver does not need to access the SCSI diagnostic commands. Commands received from the device driver through the **strategy** routine of the adapter are processed from a queue. Once the command has completed, the device driver is notified through the **iodone** kernel service.

SCSI Adapter/Device Interface

The AIX SCSI adapter device driver does not contain the **ddread** and **ddwrite** entry points, but does contain the **ddconfig**, **ddopen**, **ddclose**, **dddump**, and **ddioctl** entry points.

Therefore, the adapter device driver's entry in the kernel devsw table contains only those entries plus an additional **ddstrategy** entry point. This **ddstrategy** routine is the path that the SCSI device driver uses to pass commands to the device driver. Access to these entry points is possible through the following kernel services:

- **fp_open**
- **fp_close**
- **devdump**
- **fp_ioctl**
- **devstrat**

The SCSI adapter is accessed by the device driver through the **/dev/scsi#** special files, where # indicates ascending numbers 0,1, 2, and so on. The adapter is designed so that multiple devices on the same adapter can be accessed at the same time.

For additional information on spanned and gathered write commands, see "Understanding the Execution of Initiator I/O Requests" in *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts*.

sc_buf Structure

The I/O requests made from the SCSI device driver to the SCSI adapter device driver are completed throught the use of the **sc_buf** structure, which is defined in the **/usr/include/sys/scsi.h** header file. This structure, which is similar to the **buf** structure in other drivers, is passed between the two SCSI subsystem drivers through the **strategy** routine. The following is a brief description of the fields contained in the **sc_buf** structure:

struct buf bufstruct
: This structure is a copy of the standard **buf** structure used for the I/O request that is defined in the **/usr/include/sys/buf.h** header file. Note that the `b_work` field in the **buf** structure is reserved for use by the SCSI adapter device driver.

struct buf *bp Contains a pointer to the original buffer structure used by the process calling the SCSI device driver. It can be a pointer to a list of SCSI spanned data transfer commands or it can contain a value of NULL. A NULL value indicates that no list exists and all required information is contained in the `bufstruct` field (see the preceding paragraph). A non-NULL value also requires the `resvd1` field of the **sc_buf** structure to be NULL.

uint resvd1 This field is usually set to NULL although it can contain a pointer to a **uio** structure which is used in gathered writes.

uint resvd2 Reserved (should be set to zero).

uint resvd3 Reserved (should be set to zero).

uint resvd4 Reserved (should be set to zero).

uint timeout_value
: Contains the amount of time, in seconds, to be used in waiting for the completion of the command before it times out. A zero value indicates no timeout should be used.

uchar status_validity

This field can contain the SC_SCSI_ERROR bit flag that indicates that the `scsi_status` field return code is valid or the SC_ADAPTER_ERROR bit flag which indicates that the general_card_status return code is valid. There are three cases when considering the values of the return codes:

- If the `sc_buf.bufstruct.b_flag` field has the B_ERROR flag set, then the status field contains a valid **errno** value. If the `b_error` field has the value ENXIO, then the command needs to be restarted or the SCSI device driver canceled the request. If the `b_error` field has the value EIO, then the `status_validity` field indicates which status field, `scsi_status` or `general_card_status`, contains the error.

  If the `status_validity` field is zero, examine the `sc_buf.bufstruct.b_resid` field for any possible error. Note that `b_resid` can be nonzero even if no error occurred. Evaluate the nonzero value carefully to ensure that it is a proper error value.

- If the `sc_buf.bufstruct.b_flag` field does not have the B_ERROR flag set, then no error is being reported. However, you must still examine the `b_resid` field to determine if an error has actually occurred. If an error has occurred, it is up to the SCSI device driver to recover since device queues are not stopped and future commands can still be sent to the adapter and driver.

- If the `sc_buf.bufstruct.b_flag` field has the B_ERROR flag set, then the device queue has been halted. To recover or continue after the error, the `sc_buf.flags` field must have the SC_RESUME bit set in the first **sc_buf** structure.

uchar scsi_status

This field is valid whenever the correct bit is set in the `status_validity` field. The `sc_buf.bufstruct.b_error` field should also contain the value EIO whenever the `scsi_status` field is valid. The various valid values are shown and defined in the **/usr/include/sys/scsi.h** header file.

uchar general_card_status

This field is valid whenever the correct bit is set in the `status_validity` field. The `sc_buf.bufstruct.b_error` field should also contain the value EIO whenever the `scsi_status` field is valid. The various valid values are shown and defined in the **/usr/include/sys/scsi.h** header file.

The *general_card_status* bit is set in the `status_validity` field whenever the SCSI adapter device driver encounters an unrecoverable error. Recovered errors are those that have been corrected and logged by the adapter device driver. The SCSI adapter device driver logs both bus and adapter-related errors.

If an error is detected after a command has reached a device, it is the responsibility of the device driver to attempt recovery and log the error. Of the values shown in **/usr/include/sys/scsi.h**, the device driver should handle:

- SC_SCSI_BUS_FAULT
- SC_CMD_TIMEOUT
- SC_NO_DEVICE_RESPONSE

The SC_SCSI_BUS_FAULT error should be handled by the device driver and not the adapter device driver since this can be caused by a protocol or hardware failure.

uchar adap_q_status
This field is used to indicate that the adapter did not clear the device queue on a failure. The adapter will set this field to SC_DID_NOT_CLEAR_Q to indicate this condition. For example, this flag is returned if a check condition occurs while a command is being queued to the device.

uchar lun This field should contain the LUN of the target device. Note that if the LUN is greater than 7, this field should contain the LUN value and the `lun` field in the **scsi_cmd** structure should be 0.

uint resvd7 Reserved (should be set to zero).

uchar q_tag_msg
This field is used when the SCSI device supports command tag queueing. It should be set to 0 if the device does not support queueing. See the **/usr/include/sys/scsi.h** file for a list and explanation of the valid values for this field.

uchar flags This field contains various flags to instruct the adapter driver on how to process the transfer request. See the **/usr/include/sys/scsi.h** file for a list and explanation of the valid values for this field.

Adapter/Device Driver Intercommunication

In a typical request to the device driver, a call is first made to the device driver **strategy** routine, which takes care of any necessary queueing. This **strategy** routine then calls the device driver's **start** routine, which fills in the **sc_buf** structure and calls the adapter device driver's **strategy** routine through the **devstrat** kernel service.

The adapter's **strategy** routine validates all of the information contained in the **sc_buf** structure and also performs any necessary queueing of the transaction request. If no queueing is necessary, the adapter's **start** subroutine is called.

When an interrupt occurs, the SCSI adapter **interrupt** routine fills in the `status_validity` field and the appropriate `scsi_status` or `general_card_status` field of the **sc_buf** structure. The `bufstruct.b_resid` field is also filled in with the value of nontransferred bytes. The adapter's **interrupt** routine then passes this newly filled in **sc_buf** structure to the **iodone** kernel service which then signals the SCSI device driver's **iodone** subroutine. The adapter's **start** routine is also called from the i**nterrupt** routine to process any additional transactions on the queue.

The device driver's **iodone** routine should then process all of the applicable fields in the queued **sc_buf** structure for any errors and attempt error recovery if necessary. The device driver should then dequeue the **sc_buf** structure and then pass a pointer to the structure back to the **iodone** kernel service so that it can notify the originator of the request.

SCSI Adapter Device Driver Routines

This section describes the following routines:

- **config**
- **open**
- **close**
- **openx**
- **strategy**
- **ioctl**

config

The **config** routine performs all of the processing needed to configure, unconfigure, and read Vital Product Data for the SCSI adapter. When this routine is called to configure an adapter, it performs the required checks and building of data structures needed to prepare the adapter for the processing of requests.

When asked to unconfigure or terminate an adapter, this routine deallocates any structures defined for the adapter and marks the adapter as unconfigured. This routine can also be called to return the Vital Product Data (VPD) for the adapter, which contains information that is used to identify the serial number, change level, or part number of the adapter.

open

The **open** routine establishes a connection between a special file and a file descriptor. This file descriptor is the link to the special file that is the access point to a device and is used by all subsequent calls to perform I/O requests to the device. Interrupts are enabled and any data structures needed by the adapter driver are also initialized.

close

The **close** routine marks the adapter as closed and disables all future interrupts, which causes the driver to reject all future requests to this adapter.

openx

The **openx** routine allows a process with the proper authority to open the adapter in diagnostic mode. If the adapter is already open in either normal or diagnostic mode, the **openx** subroutine has a return value of –1. Improper authority results in an **errno** value of EPERM,while an already open error results in an errno value of EACCES. If the adapter is in diagnostic mode, only the **close** and **ioctl** routines are allowed. All other routines return a value of –1 and an errno value of EACCES.

While in diagnostics mode, the adapter can run diagnostics, run wrap tests, and download microcode. The **openx** routine is called with an *ext* parameter that contains the adapter mode and the SC_DIAGNOSTIC value, both of which are defined in the **sys/scsi.h** header file.

strategy

The **strategy** routine is the link between the device driver and the SCSI adapter device driver for all normal I/O requests. Whenever the SCSI device driver receives a call, it builds an **sc_buf** structure with the correct parameters and then passes it to this routine, which in turn queues up the request if necessary. Each request on the pending queue is then processed by building the necessary SCSI commands required to carry out the request. When the command has completed, the SCSI device driver is notified through the **iodone** kernel service.

ioctl

The **ioctl** routine allows various diagnostic and nondiagnostic adapter operations. Operations include the following:

- **IOCINFO**
- **SCIOSTART**
- **SCIOEVENT**
- **SCIOSTOP**
- **SCIOINQU**
- **SCIOSTUNIT**
- **SCIOTUR**
- **SCIOREAD**
- **SCIORESET**
- **SCIOHALT**
- **SCIODIAG**
- **SCIOTRAM**
- **SCIODNLD**
- **SCIOSTARTTGT**
- **SCIOSTOPTGT**

SCSI Adapter ioctl Operations

This section describes the following ioctl operations:

- **IOCINFO**
- **SCIOSTART**
- **SCIOSTOP**
- **SCIOINQU**
- **SCIOSTUNIT**
- **SCIOTUR**
- **SCIORESET**
- **SCIOHALT**
- **SCIODIAG**
- **SCIOTRAM**
- **SCIODNLD**

IOCINFO

This operation allows a SCSI device driver to obtain important information about a SCSI adapter, including the card's SCSI ID and the maximum data transfer size in bytes. By knowing the maximum data transfer size, a SCSI device driver can control several different devices on several different adapters. This operation returns a **devinfo** structure as defined in the **sys/devinfo.h** header file with the device type **DD_BUS** and subtype **DS_SCSI**. The following is an example of a call to obtain the information:

```
rc = fp_ioctl(fp, IOCINFO, &infostruct, NULL);
```

where *fp* is a pointer to a file structure and *infostruct* is a **devinfo** structure. A non-zero *rc* value indicates an error. Note that the **devinfo** structure is a union of several structures and that **scsi** is the structure that applies to the adapter.

For example, the maximum transfer size value is contained in the variable *infostruct.un.scsi.max_transfer* and the card ID is contained in *infostruct.un.scsi.card_scsi_id*.

SCIOSTART

This operation opens a logical path to the SCSI device and causes the SCSI adapter device driver to allocate and initialize all of the data areas needed for the SCSI device. The SCIOSTOP operation should be issued when those data areas are no longer needed. This operation should be issued before any nondiagnostic operation except for IOCINFO. The following is a typical call:

```
rc = fp_ioctl(fp, SCIOSTART, idlun, NULL);
```

where *fp* is a pointer to a file structure and *idlun* is a type int value that contains the SCSI and Logical Unit Number (LUN) ID values of the device to be started. The least significant byte contains the LUN, the next least significant byte contains the SCSI ID, and the most significant two bytes should be set to zero.

A nonzero return value indicates an error has occurred and all operations to this SCSI/LUN pair should cease since the device is either already started or failed the start operation. Possible errno values are **EIO**, **EINVAL**, or **EACCES**.

EIO	The command could not complete due to a system error.
EINVAL	Either the Logical Unit Number (LUN) ID or SCSI ID is invalid, or the adapter is already open.
EACCES	The adapter is not in normal mode.

SCIOSTOP

This operation closes a logical path to the SCSI device and causes the SCSI adapter device driver to deallocate all data areas that were allocated by the SCIOSTART operation. This operation should only be issued after a successful SCIOSTART operation to a device. The following is a typical call:

```
rc = fp_ioctl(fp, SCIOSTOP, idlun, NULL);
```

where *fp* is a pointer to a file structure and *idlun* is a type int value that contains the SCSI and LUN ID values of the device to be stopped. The least significant byte contains the LUN, the next least significant byte contains the SCSI ID, and the upper two bytes should be set to zero. A non-zero return value indicates an error has occurred. Possible errno values are **EIO** and **EINVAL**.

EIO	An unrecoverable system error has occurred.
EINVAL	The adapter was not in open mode.

This operation requires **SCIOSTART** to be run first.

SCIOINQU

This operation issues an inquiry command to a SCSI device and is used to aid in SCSI device configuration. The following is a typical call:

```
rc = ioctl(adapter, SCIOINQU, &inquiry_block);
```

where *adapter* is a file descriptor and *inquiry_block* is a **sc_inquiry** structure as defined in the **/usr/include/sys/scsi.h** header file. The SCSI ID and LUN should be placed in the **sc_inquiry** parameter block. The SC_ASYNC flag should not be set on the first call to this operation and should only be set if a bus fault has occurred. Possible errno values are **EIO**, **EFAULT**, **EINVAL**, **EACCES**, **ENOMEM**, **ETIMEDOUT**, **ENODEV**, and **ENOCONNECT**.

EIO	A system error has occurred. Consider retrying the operation several times, because another attempt may be successful.
EFAULT	A user process copy has failed.
EINVAL	The device is not opened.
EACCES	The adapter is in diagnostics mode.
ENOMEM	A memory request has failed.

ETIMEDOUT The command has timed out. Consider retrying the operation several times, because another attempt may be successful.

ENODEV The device is not responding. Possibly no LUNs exist on the present SCSI ID.

ENOCONNECT A bus fault has occurred and the operation should be retried with the SC_ASYNC flag set in the **sc_inquiry** structure. In the case of multiple retries, this flag should be set only on the last retry.

This operation requires **SCIOSTART** to be run first.

SCIOSTUNIT

This operation issues a start unit command to a SCSI device and is used to aid in SCSI device configuration. The following is a typical call:

```
rc = ioctl(adapter, SCIOSTUNIT, &start_block);
```

where *adapter* is a file descriptor and *start_block* is a **sc_startunit** structure as defined in the /**usr**/**include**/**sys**/**scsi.h** header file. The SCSI ID and LUN should be placed in the sc_startunit parameter block. The `start_flag` field designates the start option, which when set to true, makes the device available for use. When this field is set to false, the device is stopped.

The SC_ASYNC flag should not be set on the first call to this operation and should only be set if a bus fault has occurred. The `immed_flag` field allows overlapping start operations to several devices on the SCSI bus. When this field is set to false, status is returned only when the operation has completed. When this field is set to true, status is returned as soon as the device receives the command. The **SCIOTUR** operation can then be issued to check on completion of the operation on a particular device.

Note that when the SCSI adapter is allowed to issue simultaneous start operations, it is important that a delay of 10 seconds be allowed between successive **SCIOSTUNIT** operations to devices sharing a common power supply since damage to the system or devices can occur if this precaution is not followed. Possible error values are **EIO**, **EFAULT**, **EINVAL**, **EACCES**, **ENOMEM**, **ETIMEDOUT**, **ENODEV**, and **ENOCONNECT**.

EIO A system error has occurred. Consider retrying the operation several times, because another attempt may be successful.

EFAULT A user process copy has failed.

EINVAL The device is not opened.

EACCES The adapter is in diagnostics mode.

ENOMEM A memory request has failed.

ETIMEDOUT The command has timed out. Consider retrying the operation several times, because another attempt may be successful.

ENODEV The device is not responding. Possibly no LUNs exist on the present SCSI ID.

ENOCONNECT A bus fault has occurred. Try the operation again with the SC_ASYNC flag set in the **sc_inquiry** structure. In the case of multiple retries, this flag should be set only on the last retry.

This operation requires **SCIOSTART** to be run first.

SCIOTUR

This operation issues a SCSI Test Unit Ready command to an adapter and aids in SCSI device configuration. The following is a typical call:

```
rc = ioctl(adapter, SCIOTUR, &ready_struct);
```

where *adapter* is a file descriptor and *ready_struct* is a **sc_ready** structure as defined in the /**usr**/**include**/**sys**/**scsi.h** header file. The SCSI ID and LUN should be placed in the sc_ready parameter block. The SC_ASYNC flag should not be set on the first call to this operation and should only be set if a bus fault has occurred. The status of the device can be determined by evaluating the two output fields: `status_validity` and `scsi_status`. Possible errno values are **EIO**, **EFAULT**, **EINVAL**, **EACCES**, **ENOMEM**, **ETIMEDOUT**, **ENODEV**, and **ENOCONNECT**.

EIO A system error has occurred. Consider retrying the operation several (around three) times, because another attempt may be successful. If an EIO error occurs and the `status_validity` field is set to SC_SCSI_ERROR, then the `scsi_status` field has a valid value and should be inspected.

If the `status_validity` field is zero and remains so on successive retries, then an unrecoverable error has occurred with the device.

If the `status_validity` field is SC_SCSI_ERROR and the `scsi_status` field contains a Check Condition status, then the **SCIOTUR** operation should be retried after several seconds.

If after successive retries, the Check Condition status remains, the device should be considered inoperable.

EFAULT A user process copy has failed.

EINVAL The device is not opened.

EACCES The adapter is in diagnostics mode.

ENOMEM A memory request has failed.

ETIMEDOUT The command has timed out. Consider retrying the operation several times, because another attempt may be successful.

ENODEV The device is not responding and possibly no LUNs exist on the present SCSI ID.

ENOCONNECT A bus fault has occurred and the operation should be retried with the SC_ASYNC flag set in the **sc_inquiry** structure. In the case of multiple retries, this flag should be set only on the last retry.

This operation requires **SCIOSTART** to be run first.

SCIORESET

This operation causes a SCSI device to release all reservations, clear all current commands, and return to an initial state by issuing a Bus Device Reset (BDR) to all LUNs associated with the specified SCSI ID. A SCSI reserve command should be issued after the **SCIORESET** operation to prevent other initiators from claiming the device. Note that because a certain amount of time exists between a reset and reserve command, it is still possible for another initiator to successfully reserve a particular device. The following is a typical call:

```
rc = fp_ioctl(fp, SCIORESET, idlun, NULL);
```

where *fp* is a pointer to a file structure and *idlun* is a type int value that contains the SCSI and LUN ID values of the device to be stopped. The least significant byte contains the LUN, the next least significant byte contains the SCSI ID, and the upper two bytes should be set to zero. A nonzero return value indicates an error has occurred. Possible errno values are **EIO**, **EINVAL**, **EACCES**, and **ETIMEDOUT**.

EIO An unrecoverable system error has occurred.

EINVAL The device is not opened.

EACCES The adapter is in diagnostics mode.

ETIMEDOUT The operation did not complete before the time-out value was exceeded.

This operation requires **SCIOSTART** to be run first.

SCIOHALT

This operation stops the current command of the selected device, clears the command queue of any pending commands, and brings the device to a halted state. The SCSI adapter sends a SCSI abort message to the device and is usually used by the SCSI device driver to abort the current operation instead of allowing it to complete or time out.

After the **SCIOHALT** operation is sent, the device driver must set the SC_RESUME flag in the next **sc_buf** structure sent to the adapter device driver, or all subsequent **sc_buf** structures sent are ignored.

The SCSI adapter also performs normal error recovery procedures during this command which include issuing a SCSI bus reset in response to a SCSI bus hang. The following is a typical call:

```
rc = fp_ioctl(fp, SCIOHALT, idlun, NULL);
```

where *fp* is a pointer to a file structure and *idlun* is a type int value that contains the SCSI and LUN ID values of the device to be stopped. The least significant byte contains the LUN, the next least significant byte contains the SCSI ID, and the upper two bytes should be set to zero. A nonzero return value indicates an error has occurred. Possible errno values are **EIO**, **EINVAL**, **EACCES**, and **ETIMEDOUT**.

EIO An unrecoverable system error has occurred.

EINVAL The device is not opened.

EACCES The adapter is in diagnostics mode.

ETIMEDOUT The operation did not complete before the time-out value was exceeded.

This operation requires **SCIOSTART** to be run first.

SCIODIAG

This command is most commonly used by a SCSI adapter diagnostic program. The **SCIODIAG** operation allows the SCSI adapter to run various diagnostic commands, which include:

- Internal Diagnostics Test
- SCSI Wrap Test
- Read/Write Register Test
- POS Register Test

These diagnostics options are defined, along with the **sc_card_diag** structure which is used in the call, in the **/usr/include/sys/scsi.h** header file. Any errors detected by the diagnostics are returned in the same **sc_card_diag** structure. Refer to the header file for a definition of the returned values.

Whenever an error is detected, an EFAULT errno value is returned along with the appropriate error statuses in the **sc_card_diag** structure. When the ENOMSG value is received, no information is provided in the error status fields.

Because this operation attempts no retries or error recovery, no error logging is provided. The following is a typical call:

```
rc = ioctl(adapter, SCIODIAG, &diag_struct);
```

where *adapter* is a file descriptor and *diag_struct* is a **sc_card_diag** structure. Possible errno values are **EIO**, **EFAULT**, **EINVAL**, **EACCES**, **ENOMSG**, and **ETIMEDOUT**.

EIO	An unrecoverable system error has occurred.
EFAULT	A user process copy has failed or diagnostics has failed without completing all tests.
EINVAL	An invalid diagnostic command was passed.
EACCES	The adapter is not in diagnostics mode.
ENOMSG	The diagnostic command has completed with errors.
ETIMEDOUT	The operation did not complete before the time-out value was exceeded.

This operation requires the adapter be in diagnostic mode.

SCIOTRAM

This operation is designed to test SCSI adapter RAM. However, it is currently not supported and therefore, returns an errno value of ENXIO if called.

SCIODNLD

This operation downloads microcode to the SCSI adapter and is used in configuring the SCSI adapter. This command can also be used to query for the current version of the microcode from the adapter. The following is a typical call:

```
rc = ioctl(adapter, SCIODNLD, &dnld_struct);
```

where *adapter* is a file descriptor and *dnld_struct* is a **sc_download** structure. Possible errno values are **EIO**, **EFAULT**, **EINVAL**, **EACCES**, **ENOMEM**, and **ETIMEDOUT**.

EIO An unrecoverable system error has occurred or there is a checksum error with the microcode. If the adapter has been opened in diagnostics mode, this error is logged in the system error log. If the adapter has onboard microcode, it may still function properly.

EFAULT A user copy has failed or a severe I/O error has occurred during the download and all subsequent commmands to this adapter should cease. If the adapter has been opened in diagnostics mode, this error is logged in the system error log.

EINVAL An invalid input parameter was passed.

ENOMEM A request for memory failed.

ETIMEDOUT The operation did not complete before the time-out value was exceeded.

SCSI Device Driver Routines

A SCSI device driver should contain the following routines:

- **config**
- **ioctl**
- **open**
- **close**
- **read**
- **write**

Additional routines that you may need are the **strategy, iodone**, and **dump** routines. A **dump** routine is needed if the device is to be used as the receiver of a system dump. A **strategy** routine is needed in the case of block I/O device drivers that handle lists of **buf** structures.

Because the AIX operating system allows device drivers to be paged out of memory, certain functions and data structures of the driver must be guaranteed to be in memory to avoid page faults while running in the interrupt environment.This is required because disabling interrupts to any level will prevent the system from paging any pages in or out.

Routines that are called from an interrupt handler run in the interrupt environment while those that are called from a kernel or user process run in the process environment. Routines that run in the process environment can be interrupted by those running in the interrupt environment and can be preempted by processes with a higher process priority.

Routines running in the interrupt environment can only be interrupted by those running at a higher interrupt priority or by exceptions. In order for interrupt environment routines to avoid causing page faults, all code and data accessed in the interrupt environment must be pinned. Kernel services that deal with the pinning of code are **pin**, **pinu**, and **pincode**.

A function name is passed to the **pincode** kernel service when using it to pin driver code. However, the entire module that contains the function is pinned, not just the function itself. In the case of large device drivers, this is a wasteful use of memory. To get around this restriction, split the driver into two halves:

- a top half
- a bottom half

The top half contains preemptable routines that run in the process environment while the bottom half contains routines that run in the interrupt environment.

You can compile the two halves so that they are cross-linked. This allows one half of the driver to be automatically loaded together with the other half when it is loaded. The bottom (pinned) half should not have any dependencies on the top (unpinned) half, so you must carefully plan the two halves to avoid this condition, while at the same time minimizing wasted memory.

Usually, top-half routines include the **config**, **ioctl**, **open**, **close**, **read**, and **write** routines. Routines that usually reside in the bottom half are the **strategy**, **dump**, and **iodone** routines. Put any routines that cannot be paged out in the bottom half of the device driver.

In the case of read-only devices, such as CD-ROM, you do not need a **write** routine.

Top-Half Routines

This section discusses the following routines:

- **config**
- **ioctl**
- **open**
- **close**
- **read**
- **write**

config

This routine is commonly used when configuring, unconfiguring, or changing attributes of the device. It is called by various configuration methods which include the configure, unconfigure, and change methods. This routine can also be called to return the Vital Product Data (VPD) of the device. The following is a typical call to this subroutine from a configuration method:

```
sysconfig(SYS_CFGDD, &cfg, sizeof(struct cfg_dd));
```

where **sysconfig** is a subroutine and *cfg* is a **cfg_dd** structure which is defined in /**usr**/**include**/**sys**/**sysconfig.h**. The `cmd` field of the **cfg_dd** structure should contain one of the following parameters:

- CFG_INIT
- CFG_TERM
- CFG_QVPD

If the routine is called with the CFG_INIT parameter, perform simple error checking to ensure that the driver, as well as the device, is in the correct state. Any required data structures should also be allocated and initialized with the values contained in the Device Dependent Structure (DDS). The DDS should have been correctly built by the configuration method and passed to the config routine through the **cfg_dd** structure which contains a field that points to the DDS. After all data structures have been properly initialized, the device switch table entry should be built and then entered using the **devswadd** kernel service.

If called with the CFG_TERM parameter, the routine should first check that the driver is in the correct state and that the device is closed before deallocating associated data structures, removing the driver from the device switch table, and changing the status of the device to CLOSED in the device driver.

If the routine is called with the CFG_QVPD parameter, the driver should fill in the passed **uio** structure with the appropriate information about the device and the driver and then return this structure to the calling routine.

ioctl

This routine handles any I/O access to the SCSI device other than that which is handled by the **read** and **write** routines. Any diagnostic capabilities and the standard **IOCINFO** operation should also be handled by this routine.

Typical I/O requests are those that reset the device, obtain certain configuration parameters, or perform various basic accesses to the device. This routine can also perform various diagnostic functions once the device is opened in diagnostic mode. The **IOCINFO** operation is used to return information specific to the device as well as statistics about the usage of the device.

open

This routine opens a SCSI device by initializing any data structures, pinning the bottom half of the driver code, registering and initializing and interrupt handlers, setting up any DMA channels, and preparing any needed timers.

If the current open is the first open to the device, any global data structures as well as the bottom half of the driver should be pinned using the **pincode** kernel service. The pinning of the code and global data structures should occur before the device is able to generate interrupts since the interrupt handler is usually placed in the bottom half of the driver.

The SCSI adapter should then be opened with the **fp_open** kernel service. Once this operation completes, a **SCIOSTART** should then be issued for the device. If a forced open was attempted on the device, then a bus device reset (BDR) must also be issued through the **SCIORESET** operation. If the open operation was a diagnostic open, then processing is usually complete at this point since all that is required for a diagnostic open is that the appropriate data structures are pinned and the adapter is opened and started for the device.

If the open is a normal open, then an **SCIOTUR** operation should be issued to test for the readiness of the SCSI device. An **IOCINFO** ioctl call should then be made to obtain any necessary operating parameters for the driver.

The open routine should serialize its operation using simple locks. This prevents more than one application from modifying driver data structures. Unlock the resource when the driver has completed its processing to allow other applications to open SCSI devices.

Note that if any of the commands during the open operation fail, the device driver should clean up before exiting with an error. This includes freeing any allocated memory, unpinning any pinned code, and releasing any locks.

close

This routine closes the SCSI device by deallocating any data structures associated with the device as well as unpinning any code or data. The adapter driver is also stopped and any timers are also released.

Serialize the **close** routine so that no more than one application can modify the data structures of a driver. Use simple lock kernel services to accomplish this.

If the device was opened in normal mode and the SC_RETAIN_RESERVE flag was not set, then a release must be performed on the device to allow other initiators to reserve the device. If the reservation is to be retained, then the release should not be performed.

A **SCIOSTOP** operation should now be issued to the adapter driver which should also be closed using the **fp_close** kernel service.

read

This routine reads data from a SCSI device and returns it to the calling process through a **uio** structure.

The driver must perform any parameter validation, command building, and cross memory descriptor processing that might be necessry before calling the SCSI adapter strategy routine through **devstrat**. For block device drivers, if the read routine is merely a raw interface, the **uphysio** routine should be called if the request falls on block boundaries. The **uphysio** routine will end up calling the device driver's strategy routine. If the request does not fall on a block boundary, the driver should then break up the request into blocks. Odd sized transfers can then be handled by using **devstrat** to call the strategy routine after the **sc_buf** structure has been correctly built.

write

This routine writes data received through a **uio** structure from the calling process to a SCSI device.

The write routine should be very similar to the read routine described above in the type of prcessing that is required.

Bottom-Half Routines

This section discusses the following routines:

- **strategy**
- **dump**

strategy

Note: This routine is only present in drivers for block SCSI devices. This routine accepts a linked list of **buf** structures, processes them, and determines the proper SCSI command to send to the device to perform the required operation. The **buf** structure is defined in the /**usr**/**include**/**sys**/**buf.h** header file.

This routine should also determine if requests made in successive **buf** structures can be consolidated into one SCSI command. The request is placed on the I/O request queue for future processing by the device driver. Once a request is ready to be sent out to the device, the adapter's strategy routine should be called via the **devstrat** kernel routine. The adapter's strategy routine takes a pointer to an **sc_buf** structure as its only parameter.

After the I/O operation has completed, the user-level calling routine's **iodone** routine is called through the **iodone** kernel service. The caller's **iodone** routine should have been passed in the `b_iodone` field of the **buf** structure.

dump

This routine is needed if the device is to be used as a possible recipient of a system dump. The dump routine should first disable interrupts to the INTIODONE level which are naturally re-enabled at the end of the routine. Also, there are several commands, passed through the `cmd` parameter, that the routine must handle.

For a DUMPINIT command, this routine should just call the adapter dump routine through the **devdump** system call with a DUMPINIT command.

For a DUMPSTART command, this routine should just call the adapter dump routine through the **devdump** system call with a DUMPSTART command.

For a DUMPQUERY command, this routine should just call the adapter dump routine through the **devdump** system call with a DUMPQUERY command and then fill in the **dmp_query** structure passed in through the `arg` parameter with the appropriate information.

For a DUMPWRITE command, this routine should fill out an **sc_buf** structure for each i**ovec** structure passed in through the uio parameter. A SCSI WRITE command should be constructed into each of the **sc_buf** structures and the adapter's dump routine should then be called through the **devdump** system call with a DUMPWRITE command. This should be repeated until all the **iovec** structures have been processed.

For a DUMPEND command, this routine should just call the adapter dump routine through the **devdump** system call with a DUMPEND command.

For a DUMPTERM command, this routine should just call the adapter dump routine through the **devdump** system call with a DUMPTERM command.

PVIDs

If a SCSI device is intended to contain a JFS filesystem, it must first meet two requirements. The first is that even though the device is not a hard disk, the driver must respond as such when it is issued an IOCINFO **ioctl** call. The device type must be either DD_DISK or DD_SCDISK as defined in the header file **/usr/include/sys/devinfo.h**.

The second requirement is that the device must contain a Physical Volume Identifier (PVID) so that it can be uniquely distinguished from other media on the system that contain filesystems. The PVID is used by the system to keep track of media even if it is moved from one SCSI ID to another or from one adapter to another. This allows the Logical Volume Manager (LVM) to maintain the consistency of volume groups (VGs) and logical volumes (LVs) that span several physical storage devices.

Normally, once a PVID is written, it remains with the device until it is overwritten or somehow damaged. This implies that a PVID only needs to be written once for a newly added device. New system-supported hard disks will have a PVID assigned to them the first time they are configured on a system by the configure method for hard disks. The system accomplishes this by first performing a read of the IPL record area from each disk detected on the system. If this area contains no PVID (the PVID value is 0), one is created and written out to the IPL record area. The PVID is created by the following scheme:

```
  #define makehex(x) "0123456789abcdef"[x&15]

  struct unique_id           unique_id;
  struct utsname             uname_buf;
  long                       machine_id;
  struct timestruc_t         cur_time;
  int                        i;
  char                       pvidstr[33];
  char                       bdevice[64];
  int                        fd;
  IPL_REC                    clearipl;
  IPL_REC                    ipl_rec;
  off_t                      offset;

  bzero((caddr_t) &unique_id, sizeof (struct unique_id));

  if (gettimer (TIMEOFDAY, &cur_time))
        exit (-1);

  if (uname(&uname_buf))
        exit (-1);

  machine_id = 0;
  sscanf(uname_buf.machine, "%8x", &machine_id);

/* Note that words 3 and 4 remain 0 */
  unique_id->word1 = machine_id;
  unique_id->word2 = cur_time.tv_sec*1000 + cur_time.tv_nsec/1000000;

for(i=0;i<32;i++) {
        if (i&1)
                pvidstr[i] = makehex(unique_id[i/2]);
        else
                pvidstr[i] = makehex(unique_id[i/2]>>4);
  }
  pvidstr[32] = '\0';

 /* lname is the name of the disk */
  sprintf(bdevice,"/dev/r%s",lname);
  fd = open(bdevice, O_RDWR);
  if (fd < 0)
        exit (-1);

offset = lseek(fd, PSN_IPL_REC, 0);
  if (offset < 0)
        exit (-1);

 if (read(fd, &ipl_rec, sizeof(ipl_rec)) < 0)
        exit (-1);

  if (ipl_rec.IPL_record_id != (unsigned int) IPLRECID) {
        /*
         * Boot record does not exist on disk yet.
         */
                ipl_rec = clearipl;
                ipl_rec.IPL_record_id = IPLRECID;
  }
  ipl_rec.pv_id = pvid;

offset = lseek(fd, PSN_IPL_REC, 0);
  if (offset < 0)
        exit (-1);

  if (write(fd, &ipl_rec, sizeof(ipl_rec)) < 0)
        exit (-1);

  close (fd);
```

Once the PVID has been created and stored to disk, it should also be added into the CuAt database for the disk under the `pvid` attribute.

SCSI Device Attributes

The following is a list of standard attributes that apply to the supported SCSI devices. This list contains attributes most SCSI devices should contain. It is not an exhaustive list since each device may require additional attributes, depending on the implementation of each device.

model_name The name of the SCSI device that is returned in the inquiry string when an **SCIOINQU** ioctl call is made to the device. It is usually a 16-byte character string.

maxlun The maximum allowed value of the Logical Unit Number (LUN) for this device. Currently, all supported SCSI devices have a **maxlun** value of zero. This indicates that only that particular device is allowed to occupy its SCSI ID and instructs the SCSI adapter to cease searching for other devices at other LUNs on the current SCSI ID.

pvid This attribute applies only to SCSI disks and contains the PVID value of the disk which is stored in the boot record of the disk. The value is usually a 32-bit character string and its default value is “none” as stored in the Predefined Attribute (**PdAt**) object class. Once a disk is assigned a PVID, create a Customized Attribute (**CuAt**) object class entry that contains the proper PVID value of the disk.

SCSI Configuration Methods

The configuration of SCSI devices is similar to the configuration of other devices on the system. Please refer to “Device Configuration Methods” on page 6-1 for a detailed explanation of configuration methods and how they should be written. However, there is an additional requirement that configuration methods for SCSI devices be written to take care of PVIDs (Physical Volume Identifiers) if necessary.

The configure method should try to read the PVID from the disk after the disk has been spun up and an INQUIRY command has been issued. If the disk has no PVID, then one should be created and wriiten out if the media is allowed to contain an LV (logical volume) or filesystem. For information on how to accomplish this, see “PVIDs” on page 8-17.

If a non-NULL PVID is read, either of the following situations may exist:

- The device is already known to the system.
- The device is being moved from another system.

To determine which situation exists, the config method should scan the CuAt database for a previous record of this PVID. If none is found, then the disk should be assumed to be one being moved from another system.

If a non-NULL PVID is read, then the config method should verify that the PVID stored in the database is for the same logical name as the disk currently being configured. If so, then the disk is in the same position as it was when it was last configured and the config method does not need to perform any more PVID processing.

If a non-NULL PVID is read but its matching database entry does not match the logical name of the device being currently configured, this is an indication that the device has been moved from another location on the same system. If this occurs, calling the current parent SCSI adapter’s config method will update the CuDv to correctly pair the device’s logical name with its PVID.

Chapter 9. Integrated Device Electronics (IDE) Device Drivers

The Integrated Device Electronics (IDE) subsystem has two parts:

- IDE Device Driver
- IDE Adapter Device Driver

The role of the IDE device driver is to pass information between the operating system and the IDE adapter device driver by accepting I/O requests and passing these requests to the IDE adapter device driver. The device driver should accept either character or block I/O requests, build the necessary IDE commands, and then issue these commands to the device through the IDE adapter device driver.

The IDE device driver is notified through the **iodone** kernel service once the adapter has completed the processing of the command. The device driver should then notify its calling process that the request has completed processing through the **iodone** kernel service.

IDE Adapter Device Driver Overview

Unlike most other device drivers, the IDE adapter device driver does not support the **read** and **write** subroutines. It only supports the **open**, **close**, **ioctl**, **config**, and **strategy** subroutines.

Commands received from the device driver through the **strategy** routine of the adapter are processed from a queue. Once the command has completed, the device driver is notified through the **iodone** kernel service.

IDE Adapter/Device Interface

The AIX IDE adapter device driver does not contain the **ddread** and **ddwrite** entry points, but does contain the **ddconfig**, **ddopen**, **ddclose**, **dddump**, and **ddioctl** entry points.

Therefore, the adapter device driver's entry in the kernel **devsw** table contains only those entries plus an additional **ddstrategy** entry point. This **ddstrategy** routine is the path that the IDE device driver uses to pass commands to the device driver. Access to these entry points is possible through the following kernel services:

- **fp_open**
- **fp_close**
- **devdump**
- **fp_ioctl**
- **devstrat**

The IDE adapter is accessed by the device driver through the /**dev**/**ide#** special files, where # indicates ascending numbers 0,1, 2, and so on. The adapter is designed so that multiple devices on the same adapter can be accessed at the same time.

For additional information on spanned and gathered write commands, see "Execution of I/O Requests" in *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts*.

ataide_buf Structure

The I/O requests made from the IDE device driver to the IDE adapter device driver are completed through the use of the **ataide_buf** structure defined in the **/usr/include/sys/ide.h** header file. This structure, which is similar to the **buf** structure in other drivers, is passed between the two IDE subsystem drivers through the **strategy** routine. The following is a brief description of the fields contained in the **ataide_buf** structure:

struct buf bufstruct
: This structure is a copy of the standard buf structure used for the I/O request that is defined in the **/usr/include/sys/buf.h** header file. Note that the `b_work` field in the **buf** structure is reserved for use by the IDE adapter device driver.

struct buf *bp
: Contains a pointer to the original buffer structure used by the process calling the IDE device driver. It can be a pointer to a list of IDE spanned data transfer commands or it can contain a value of NULL. A NULL value indicates that no list exists and all required information is contained in the `bufstruct` field (see the preceding paragraph).

uint timeout_value
: Contains the amount of time, in seconds, to be used in waiting for the completion of the command before it times out. A zero value indicates no time-out should be used.

uchar status_validity
: This field can contain the ATA_IDE_STATUS bit flag that indicates that the `ata.status` field return code is valid. The ATA_ERROR_VALID bit flag indicates that the `ata.errval` field contains a valid error indicator. There are three cases when considering the values of the return codes:

  - If the `ataide_buf.bufstruct.b_flag` field has the **B_ERROR** flag set, then the `ataide_buf.bufstruct.b_error` field contains a valid **errno** value. If the `b_error` field has the value **ENXIO**, then the command needs to be restarted or the IDE device driver canceled the request. If the `b_error` field has the value **EIO**, then the `status_validity` field indicates which fields, `ata.status` or `ata.errval`, are valid.

    If the `ata.status` field indicates no error, examine the `ataide_buf.bufstruct.b_resid` field for any possible error. Note that `b_resid` can be nonzero even if no error occurred. Evaluate the nonzero value carefully to ensure that it is a proper error value.

  - If the `ataide_buf.bufstruct.b_flag` field does not have the B_ERROR flag set, then no error is being reported. However, you must still examine the `b_resid` field to determine if an error has actually occurred. If an error has occurred, it is up to the IDE device driver to recover since device queues are not stopped and future commands can still be sent to the adapter and driver.

  - There is a special case when `b_resid` will be nonzero. The DMA service routine may not be able to map all virtual to real memory pages for a single DMA transfer. This may occur when sending close to the maximum amount of data that the adapter driver supports. In this case, the adapter driver transfers as much of the data that can be mapped by the DMA service. The unmapped size is returned in the `b_resid` field, and the `status_validity` field will have the ATA_IDE_DMA_NORES bit set. The IDE device driver is expected to send the data represented by the `b_resid` field in a separate request.

uchar ata.status

This field is valid whenever the ATA_IDE_STATUS bit is set in the `status_validity` field. The various valid values are shown and defined in the **/usr/include/sys/ide.h** header file.

uchar ata.errval

This field is valid whenever the ATA_ERROR_VALID bit is set in the `status_validity` field. The `ataide_buf.bufstruct.b_error` field should also contain the value EIO whenever the `ata.errval` field is valid. The various valid values are shown and defined in the **/usr/include/sys/ide.h** header file.

The ATA_ERROR_VALID bit is set in the `status_validity` field whenever the IDE adapter device driver encounters an unrecoverable error. Recovered errors are those that have been corrected and logged by the adapter device driver. The IDE adapter device driver logs both bus and adapter-related errors.

If an error is detected after a command has reached a device, it is the responsibility of the device driver to attempt recovery and log the error. Of the values shown in **/usr/include/sys/ide.h**, the device driver should handle:

- ATA_ERROR_VALID
- ATA_CMD_TIMEOUT

Adapter/Device Driver Intercommunication

In a typical request to the device driver, a call is first made to the device driver **strategy** routine, that takes care of any necessary queuing. This **strategy** routine then calls the device driver's **start** routine, which fills in the **ataide_buf** structure and calls the adapter device driver's **strategy** routine through the **devstrat** kernel service.

The adapter's **strategy** routine validates all of the information contained in the **ataide_buf** structure and also performs any necessary queuing of the transaction request. If no queuing is necessary, the adapter's **start** subroutine is called.

When an interrupt occurs, the IDE adapter **interrupt** routine fills in the `status_validity` field and the appropriate `ata.status` or `ata.errval` field of the **ataide_buf** structure. The `bufstruct.b_resid` field is also filled in with the value of nontransferred bytes. The adapter's **interrupt** routine then passes this newly filled in **ataide_buf** structure to the **iodone** kernel service which then signals the IDE device driver's **iodone** subroutine. The adapter's **start** routine is also called from the **interrupt** routine to process any additional transactions on the queue.

The device driver's **iodone** routine should then process all of the applicable fields in the queued **ataide_buf** structure for any errors and attempt error recovery if necessary. The device driver should then dequeue the **ataide_buf** structure and then pass a pointer to the structure back to the **iodone** kernel service so that it can notify the originator of the request.

IDE Adapter Device Driver Routines

This section describes the following routines:

- **config**
- **open**
- **close**
- **strategy**
- **ioctl**

config

The **config** routine performs all of the processing needed to configure, unconfigure, and read Vital Product Data for the IDE adapter. When this routine is called to configure an adapter, it performs the required checks and building of data structures needed to prepare the adapter for the processing of requests.

When asked to unconfigure or terminate an adapter, this routine deallocates any structures defined for the adapter and marks the adapter as unconfigured. This routine can also be called to return the Vital Product Data (VPD) for the adapter, which contains information that is used to identify the serial number, change level, or part number of the adapter.

open

The **open** routine establishes a connection between a special file and a file descriptor. This file descriptor is the link to the special file that is the access point to a device and is used by all subsequent calls to perform I/O requests to the device. Interrupts are enabled and any data structures needed by the adapter driver are also initialized.

close

The **close** routine marks the adapter as closed and disables all future interrupts, which causes the driver to reject all future requests to this adapter.

strategy

The **strategy** routine is the link between the device driver and the IDE adapter device driver for all normal I/O requests. Whenever the IDE device driver receives a call, it builds an **ataide_buf** structure with the correct parameters and then passes it to this routine, which in turn queues up the request if necessary. Each request on the pending queue is then processed by building the necessary IDE commands required to carry out the request. When the command has completed, the IDE device driver is notified through the **iodone** kernel service.

ioctl

The ioctl routine allows various adapter operations. Operations include the following:

- **IOCINFO**
- **IDEIOSTART**
- **IDEIOSTOP**
- **IDEIOINQU**
- **IDEIOSTUNIT**
- **IDEIOTUR**
- **IDEIOREAD**
- **IDEIORESET**
- **IDEIOIDENT**

IDE Adapter ioctl Operations

This section describes the following ioctl operations:

- **IOCINFO**
- **IDEIOSTART**
- **IDEIOSTOP**
- **IDEIOIDENT**
- **IDEIOINQU**
- **IDEIOSTUNIT**
- **IDEIOTUR**
- **IDEIORESET**

IOCINFO

This operation allows an IDE device driver to obtain important information about an IDE adapter, mainly the maximum data transfer size in bytes. By knowing the maximum data transfer size, an IDE device driver can control several different devices on several different adapters. This operation returns a **devinfo** structure as defined in the **sys/devinfo.h** header file with the device type **DD_BUS** and subtype **DS_IDE**. The following is an example of a call to obtain the information:

```
rc = fp_ioctl(fp, IOCINFO, &infostruct, NULL);
```

where `fp` is a pointer to a **file** structure and `infostruct` is a **devinfo** structure. A nonzero rc value indicates an error. Note that the **devinfo** structure is a union of several structures and that **ide** is the structure that applies to the adapter.

For example, the maximum transfer size value is contained in the variable **infostruct.un.ide.max_transfer**.

IDEIOSTART

This operation opens a logical path to the IDE device and causes the IDE adapter device driver to allocate and initialize all of the data areas needed for the IDE device. The **IDEIOSTOP** operation should be issued when those data areas are no longer needed. This operation should be issued before any operation except for **IOCINFO**. The following is a typical call:

```
rc = fp_ioctl(fp, IDEIOSTART, id, NULL);
```

where `fp` is a pointer to a **file** structure and `id` is a type int value that contains the IDE device ID value of the device to be started. The least significant byte contains the IDE device ID and the most significant three bytes should be set to zero.

A nonzero return value indicates an error has occurred and all operations to this IDE device ID should cease since the device is either already started or failed the start operation. Possible errno values are **EIO**, **EINVAL**, or **EACCES**.

EIO	The command could not complete due to a system error.
EINVAL	Either the IDE device ID is invalid, or the adapter is already open.
EACCES	The adapter is not in normal mode.

IDEIOSTOP

This operation closes a logical path to the IDE device and causes the IDE adapter device driver to deallocate all data areas that were allocated by the **IDEIOSTART** operation. This operation should only be issued after a successful **IDEIOSTART** operation to a device. The following is a typical call:

```
rc = fp_ioctl(fp, IDEIOSTOP, id, NULL);
```

where `fp` is a pointer to a **file** structure and `id` is a type int value that contains the IDE device ID value of the device to be stopped. The least significant byte contains the IDE device ID, and the upper three bytes should be set to zero. A nonzero return value indicates an error occurred. Possible errno values are **EIO** and **EINVAL**.

EIO An unrecoverable system error occurred.

EINVAL The adapter was not in open mode.

This operation requires **IDEIOSTART** to be run first.

IDEIOIDENT

This operation issues an identify device command to an IDE device and is used to aid in IDE device configuration. First an ATA identify device command is sent to the device; if this fails, an ATAPI identify device command is sent to the device. The following is a typical call:

```
rc = ioctl(adapter, IDEIOIDENT, &identify);
```

where `adapter` is a file descriptor and `identify` is an **identify_device** structure as defined in the **/usr/include/sys/ide.h** header file. The IDE device ID should be placed in the **ata** parameter block. Possible errno values are **EIO**, **EFAULT**, **EINVAL**, **EACCES**, **ENOMEM**, **ETIMEDOUT**, **ENODEV**, and **ENOCONNECT**.

EIO A system error occurred. Consider retrying the operation several times, because another attempt may be successful.

EFAULT A user process copy failed.

EINVAL The device is not opened.

ENOMEM A memory request failed.

ETIMEDOUT The command has timed out. Consider retrying the operation several times, because another attempt may be successful.

ENODEV The device is not responding. Possibly no device exists with the present IDE device ID.

ENOCONNECT
A bus fault occurred.

This operation requires **IDEIOSTART** to be run first.

IDEIOINQU

This operation issues an inquiry command to an IDE ATAPI device and is used to aid in IDE device configuration. The following is a typical call:

```
rc = ioctl(adapter, IDEIOINQU, &inquiry_block);
```

where `adapter` is a file descriptor and `inquiry_block` is an **ide_inquiry** structure as defined in the **/usr/include/sys/ide.h** header file. The IDE device ID should be placed in the **ide_inquiry** parameter block. Possible errno values are **EIO**, **EFAULT**, **EINVAL**, **ENOMEM**, **ETIMEDOUT**, **ENODEV**, and **ENOCONNECT**.

EIO A system error occurred. Consider retrying the operation everal times, because another attempt may be successful.

EFAULT A user process copy failed.

EINVAL The device is not opened.

ENOMEM A memory request failed.

ETIMEDOUT The command has timed out. Consider retrying the operation several times, because another attempt may be successful.

ENODEV The device is not responding. Possibly no device exists with the present IDE ID.

ENOCONNECT
A bus fault occurred.

This operation requires that **IDEIOSTART** run first.

IDEIOSTUNIT

This operation issues a start unit command to an IDE ATAPI device and is used to aid in IDE device configuration. The following is a typical call:

```
rc = ioctl(adapter, IDEIOSTUNIT, &start_block);
```

where `adapter` is a file descriptor and `start_block` is an **ide_startunit** structure as defined in the **/usr/include/sys/ide.h** header file. The IDE device ID should be placed in the **ide_startunit** parameter block. The `start_flag` field designates the start option, which when set to true, makes the device available for use. When this field is set to false, the device is stopped.

The `immed_flag` field is not supported; this field appears only for SCSI compatibility. Status is returned only when the operation has completed.

Possible error values are **EIO**, **EFAULT**, **EINVAL**, **ENOMEM**, **ETIMEDOUT**, **ENODEV**, and **ENOCONNECT**.

EIO A system error occurred. Consider retrying the operation several times, because another attempt may be successful.

EFAULT A user process copy failed.

EINVAL The device is not opened.

ENOMEM A memory request failed.

ETIMEDOUT The command timed out. Consider retrying the operation several times, because another attempt may be successful.

ENODEV The device is not responding.

ENOCONNECT
A bus fault occurred.

This operation requires **IDEIOSTART** to run first.

IDEIOTUR

This operation issues an IDE ATAPI Test Unit Ready command to an IDE ATAPI device and aids in IDE device configuration. The following is a typical call:

```
rc = ioctl(adapter, IDEIOTUR, &ready_struct);
```

where `adapter` is a file descriptor and **ready_struct** is an **ide_ready** structure as defined in the **/usr/include/sys/ide.h** header file. The IDE device ID should be placed in the **ide_ready** parameter block. The status of the device can be determined by evaluating the output field: `status_validity`. Possible errno values are **EIO**, **EFAULT**, **EINVAL**, **ENOMEM**, **ETIMEDOUT**, **ENODEV**, and **ENOCONNECT**.

EIO A system error occurred. Consider retrying the operation several (around three) times, because another attempt may be successful. If an **EIO** error occurs and the `status_validity` field has ATA_IDE_STATUS set, then the `ata_status` field has a valid value and should be inspected.

If the `status_validity` field is zero and remains so on successive retries, then an unrecoverable error has occurred with the device.

If the `status_validity` field has ATA_ERROR set and the `ata_status` field contains a Check Condition status, then the **IDEIOTUR** operation should be retried after several seconds.

If the Check Condition status remains after several retries, the device should be considered inoperable.

EFAULT A user process copy failed.

EINVAL The device is not opened.

ENOMEM A memory request failed.

ETIMEDOUT The command timed out. Consider retrying the operation several times, because another attempt may be successful.

ENODEV The device is not responding and possibly no device exists on the present IDE device ID.

ENOCONNECT
A bus fault occurred.

This operation requires **IDEIOSTART** to run first.

IDEIORESET

This operation causes an IDE device to clear all current commands and return to an initial state by issuing an ATAPI soft reset to the specified ATAPI device. The following is a typical call:

```
rc = fp_ioctl(fp, IDEIORESET, id, NULL);
```

where `fp` is a pointer to a **file** structure and `id` is a type int value that contains the IDE device ID value of the device to be stopped. The least significant byte contains the IDE device ID, and the upper three bytes should be set to zero. A nonzero return value indicates an error has occurred. Possible errno values are **EIO**, **EINVAL**, and **ETIMEDOUT**.

EIO An unrecoverable system error occurred.

EINVAL The device is not opened.

ETIMEDOUT The operation did not complete before the time-out value was exceeded.

This operation requires **IDEIOSTART** to run first.

IDE Device Driver Routines

An IDE device driver should contain the following routines:

- **config**
- **ioctl**
- **open**
- **close**
- **read**
- **write**

Additional routines you may need are the **strategy**, **iodone**, and **dump** routines. A **dump** routine is needed if the device is to used as the receiver of a system dump. A **strategy** routine is needed in the case of block I/O device drivers that handle lists of **buf** structures.

Because the AIX operating system allows device drivers to be paged out of memory, certain functions and data structures of the driver must be guaranteed to be in memory to avoid page faults while running in the interrupt environment. This is required because disabling interrupts to any level will prevent the system from paging any pages in or out.

Routines that are called from an interrupt handler run in the interrupt environment while those that are called from a kernel or user process run in the process environment. Routines that run in the process environment can be interrupted by those running in the interrupt environment and can be preempted by processes with a higher process priority.

Routines running in the interrupt environment can only be interrupted by those running at a higher interrupt priority or by exceptions. In order for interrupt environment routines to avoid causing page faults, all code and data accessed in the interrupt environment must be pinned. Kernel services that deal with the pinning of code are **pin**, **pinu**, and **pincode**.

A function name is passed to the **pincode** kernel service when using it to pin driver code. However, the entire module that contains the function is pinned, not just the function itself. In the case of large device drivers, this is a wasteful use of memory. To get around this restriction, split the driver into a top half and a bottom half.

The top half contains preemptable routines that run in the process environment while the bottom half contains routines that run in the interrupt environment.

You can compile the two halves so that they are cross-linked. This allows one half of the driver to be automatically loaded together with the other half when it is loaded. The bottom (pinned) half should not have any dependencies on the top (unpinned) half, so you must carefully plan the two halves to avoid this condition, while at the same time minimizing wasted memory.

Usually, top-half routines include the **config**, **ioctl**, **open**, **close**, **read**, and **write** routines. Routines that usually reside in the bottom half are the **strategy**, **dump**, and **iodone** routines. Put any routines that cannot be paged out in the bottom half of the device driver.

In the case of read-only devices, such as CD-ROM, you do not need a **write** routine.

Top-Half Routines

This section discusses the following routines:

- **config**
- **ioctl**
- **open**
- **close**
- **read**
- **write**

config

This routine is commonly used when configuring, unconfiguring, or changing attributes of the device. It is called by various configuration methods that include the **configure**, **unconfigure**, and **change** methods. This routine can also be called to return the Vital Product Data (VPD) of the device. The following is a typical call to this subroutine from a configuration method:

```
sysconfig(SYS_CFGDD, &cfg, sizeof(struct cfg_dd));
```

where `sysconfig` is a subroutine and `cfg` is a **cfg_dd** structure that is defined in **/usr/include/sys/sysconfig.h**. The `cmd` field of the **cfg_dd** structure should contain one of the following parameters:

- **CFG_INIT**
- **CFG_TERM**
- **CFG_QVPD**

If the routine is called with the **CFG_INIT** parameter, perform simple error checking to ensure that the driver, as well as the device, is in the correct state. Any required data structures should also be allocated and initialized with the values contained in the Device Dependent Structure (DDS). The DDS should have been correctly built by the configuration method and passed to the **config** routine through the **cfg_dd** structure that contains a field that points to the DDS. After all data structures have been properly initialized, the device switch table entry should be built and entered using the **devswadd** kernel service.

If called with the **CFG_TERM** parameter, the routine should first check that the driver is in the correct state and that the device is closed before deallocating associated data structures, removing the driver from the device switch table, and changing the status of the device to CLOSED in the device driver.

If the routine is called with the **CFG_QVPD** parameter, the driver should fill in the passed **uio** structure with the appropriate information about the device and the driver and then return this structure to the calling routine.

ioctl

This routine handles any I/O access to the IDE device other than that handled by the **read** and **write** routines. The standard **IOCINFO** operation should also be handled by this routine.

Typical I/O requests are those that reset the device, obtain certain configuration parameters, or perform various basic accesses to the device. The **IOCINFO** operation is used to return information specific to the device as well as statistics about the usage of the device.

open

This routine opens an IDE device by initializing any data structures, pinning the bottom half of the driver code, registering and initializing any interrupt handlers, setting up any DMA channels, and preparing any needed timers.

If the current open is the first open to the device, any global data structures as well as the bottom half of the driver should be pinned using the **pincode** kernel service. The pinning of the code and global data structures should occur before the device is able to generate interrupts since the interrupt handler is usually placed in the bottom half of the driver.

The IDE adapter should then be opened with the **fp_open** kernel service. Once this operation completes, an **IDEIOSTART** should then be issued for the device.

An **IOCINFO** ioctl call should then be made to obtain any necessary operating parameters for the driver.

The **open** routine should serialize its operation using simple locks. This prevents more than one application from modifying driver data structures. Unlock the resource when the driver has completed its processing to allow other applications to open IDE devices.

Note: If any of the commands during the **open** operation fail, the device driver should clean up before exiting with an error. This includes freeing any allocated memory, unpinning any pinned code, and releasing any locks.

close

This routine closes the IDE device by deallocating any data structures associated with the device as well as unpinning any code or data. The adapter driver is also stopped and any timers are also released.

Serialize the **close** routine so that no more than one application can modify the data structures of a driver. Use simple lock kernel services to accomplish this.

An **IDEIOSTOP** operation should now be issued to the adapter driver which should also be closed using the **fp_close** kernel service.

read

This routine reads data from an IDE device and returns it to the calling process through a **uio** structure.

The driver must perform any parameter validation, command building, and cross-memory descriptor processing that might be necessary before calling the IDE adapter **strategy** routine through **devstrat**. For block device drivers, if the **read** routine is merely a raw interface, the **uphysio** routine should be called if the request falls on block boundaries. The **uphysio** routine will end up calling the device driver's **strategy** routine. If the request does not fall on a block boundary, the driver should then break up the request into blocks. Odd-sized transfers can then be handled by using **devstrat** to call the **strategy** routine after the **ataide_buf** structure has been correctly built.

write

This routine writes data received through a **uio** structure from the calling process to an IDE device.

The **write** routine should be very similar to the **read** routine described above in the type of processing required.

Bottom-Half Routines

This section discusses the following routines:

- **strategy**
- **dump**

strategy

Note: This routine is only present in drivers for block IDE devices. This routine accepts a linked list of **buf** structures, processes them, and determines the proper IDE command to send to the device to perform the required operation. The **buf** structure is defined in the **/usr/include/sys/buf.h** header file.

This routine should also determine if requests made in successive **buf** structures can be consolidated into one IDE command. The request is placed on the I/O request queue for future processing by the device driver. Once a request is ready to be sent out to the device, the adapter's **strategy** routine should be called using the **devstrat** kernel routine. The adapter's **strategy** routine takes a pointer to an **ataide_buf** structure as its only parameter.

After the I/O operation has completed, the user-level calling routine's **iodone** routine is called through the **iodone** kernel service. The caller's **iodone** routine is passed in the `b_iodone` field of the **buf** structure.

dump

This routine is needed if the device is used as a possible recipient of a system dump. The **dump** routine should first disable interrupts to the INTIODONE level which are naturally re-enabled at the end of the routine. Also, there are several commands, passed through the **cmd** parameter, that the routine must handle:

- For a **DUMPINIT** command, this routine should call the adapter **dump** routine through the **devdump** system call with the **DUMPINIT** command.
- For a **DUMPSTART** command, this routine should call the adapter **dump** routine through the **devdump** system call with the **DUMPINIT** command.
- For a **DUMPQUERY** command, this routine should call the adapter **dump** routine through the **devdump** system call with a **DUMPQUERY** command and fill in the **dmp_query** structure passed in through the **arg** parameter with the appropriate information.
- For a **DUMPWRITE** command, this routine should fill out an **ataide_buf** structure for each **iovec** structure passed in through the **uio** parameter. An IDE WRITE command should be constructed in each of the **ataide_buf** structures and the adapter's **dump** routine should be called through the **devdump** system call with a **DUMPWRITE** command. This should be repeated until all the **iovec** structures are processed.
- For a **DUMPEND** command, this routine should call the adapter **dump** routine through the **devdump** system call with a **DUMPEND** command.
- For a **DUMPTERM** command, this routine should call the adapter **dump** routine through the **devdump** system call with a **DUMPTERM** command.

PVIDs

If an IDE device is intended to contain a JFS file system, it must first meet two requirements. The first is that even though the device is not a hard disk, the driver must respond as such when it is issued an IOCINFO ioctl call. The device type must be either DD_DISK or DD_SCDISK as defined in the header file **/usr/include/sys/devinfo.h**.

The second requirement is the device must contain a Physical Volume Identifier (PVID) so that it can be uniquely distinguished from other media on the system that contain file systems. The PVID is used by the system to keep track of media even if it is moved from one IDE connection to another or from one adapter to another. This allows the Logical Volume Manager (LVM) to maintain the consistency of volume groups (VGs) and logical volumes (LVs) that span several physical storage devices.

Normally, once a PVID is written, it remains with the device until it is overwritten or somehow damaged. This implies that a PVID only needs to be written once for a newly added device. New system-supported hard disks will have a PVID assigned to them the first time they are configured on a system by the **configure** method for hard disks. The system accomplishes this by first performing a read of the IPL record area from each disk detected on the system. If this area contains no PVID (the PVID value is 0), one is created and written out to the IPL record area. The PVID is created by the following scheme:

```
#define makehex(x) ”0123456789abcdef”[x&15]

struct unique_id                unique_id;
struct utsname                  uname_buf;
long                            machine_id;
struct timestruc_t              cur_time;
int                             i;
char                            pvidstr[33];
char                            bdevice[64];
int                             fd;
IPL_REC                         clearipl;
IPL_REC                         ipl_rec;
off_t                           offset;

bzero((caddr_t) &unique_id, sizeof (struct unique_id));

if (gettimer (TIMEOFDAY, &cur_time))
    exit (-1);

if (uname(&uname_buf))
    exit (-1);

machine_id = 0;
sscanf(uname_buf.machine, ”%8x”, &machine_id);

/* Note that words 3 and 4 remain 0 */
  unique_id->word1 = machine_id;
  unique_id->word2 = cur_time.tv_sec*1000 + cur_time.tv_nsec/1000000;

for(i=0;i<32;i++) {
      if (i&1)
              pvidstr[i] = makehex(unique_id[i/2]);
      else
              pvidstr[i] = makehex(unique_id[i/2]>4);
}
pvidstr[32] = ’0’;

/* lname is the name of the disk */
  sprintf(bdevice,”/dev/r%s”,lname);
  fd = open(bdevice, O_RDWR);
  if (fd < 0)
      exit (-1);

offset = lseek(fd, PSN_IPL_REC, 0);
  if (offset < 0)
      exit (-1);

if (read(fd, &ipl_rec, sizeof(ipl_rec)) < 0)
     exit (-1);
```

```
if (ipl_rec.IPL_record_id != (unsigned int) IPLRECID) {
    /*
     * Boot record does not exist on disk yet.
     */
            ipl_rec = clearipl;
            ipl_rec.IPL_record_id = IPLRECID;
}
ipl_rec.pv_id = pvid;

offset = lseek(fd, PSN_IPL_REC, 0);
  if (offset < 0)
      exit (-1);

if (write(fd, &ipl_rec, sizeof(ipl_rec)) < 0)
    exit (-1);

close (fd);
```

Once the PVID has been created and stored to disk, it should also be added into the CuAt database for the disk under the **pvid** attribute.

IDE Device Attributes

The following is a list of standard attributes that apply to the supported IDE devices. This list contains attributes most IDE devices should contain. It is not an exhaustive list since each device may require additional attributes, depending on the implementation of each device.

model_name The name of the IDE device that is returned in the identify device model_number string when an **IDEIOIDENT** ioctl call is made to the device. It is usually a 40-byte character string.

pvid This attribute applies only to IDE disks and contains the PVID value of the disk which is stored in the boot record of the disk. The value is usually a 32-bit character string and its default value is “none” as stored in the Predefined Attribute (PdAt) object class. Once a disk is assigned a PVID, create a Customized Attribute (CuAt) object class entry that contains the proper PVID value of the disk.

IDE Configuration Methods

The configuration of IDE devices is similar to the configuration of other devices on the system. Please refer to “Device Configuration Methods” on page 6-1 for a detailed explanation of configuration methods and how they should be written. However, there is an additional requirement that configuration methods for IDE devices be written to take care of PVIDs (Physical Volume Identifiers), if necessary.

The **configure** method should try to read the PVID from the disk after the disk has been spun up and an Identify Device command has been issued. If the disk has no PVID, then one should be created and written out if the media is allowed to contain an LV (logical volume) or file system. For information on how to accomplish this, see “PVIDs” on page 9-13.

If a non-NULL PVID is read, either of the following situations may exist:

- The device is already known to the system.
- The device is being moved from another system.

To determine which situation exists, the **config** method should scan the CuAt database for a previous record of this PVID. If none is found, then the disk should be assumed to be one being moved from another system.

If a non-NULL PVID is read, then the **config** method should verify that the PVID stored in the database is for the same logical name as the disk currently being configured. If so, then the disk is in the same position as it was when it was last configured and the **config** method does not need to perform any more PVID processing.

If a non-NULL PVID is read but its matching database entry does not match the logical name of the device being currently configured, this is an indication that the device has been moved from another location on the same system. If this occurs, calling the current parent IDE adapter’s **config** method updates the CuDv to correctly pair the device’s logical name with its PVID.

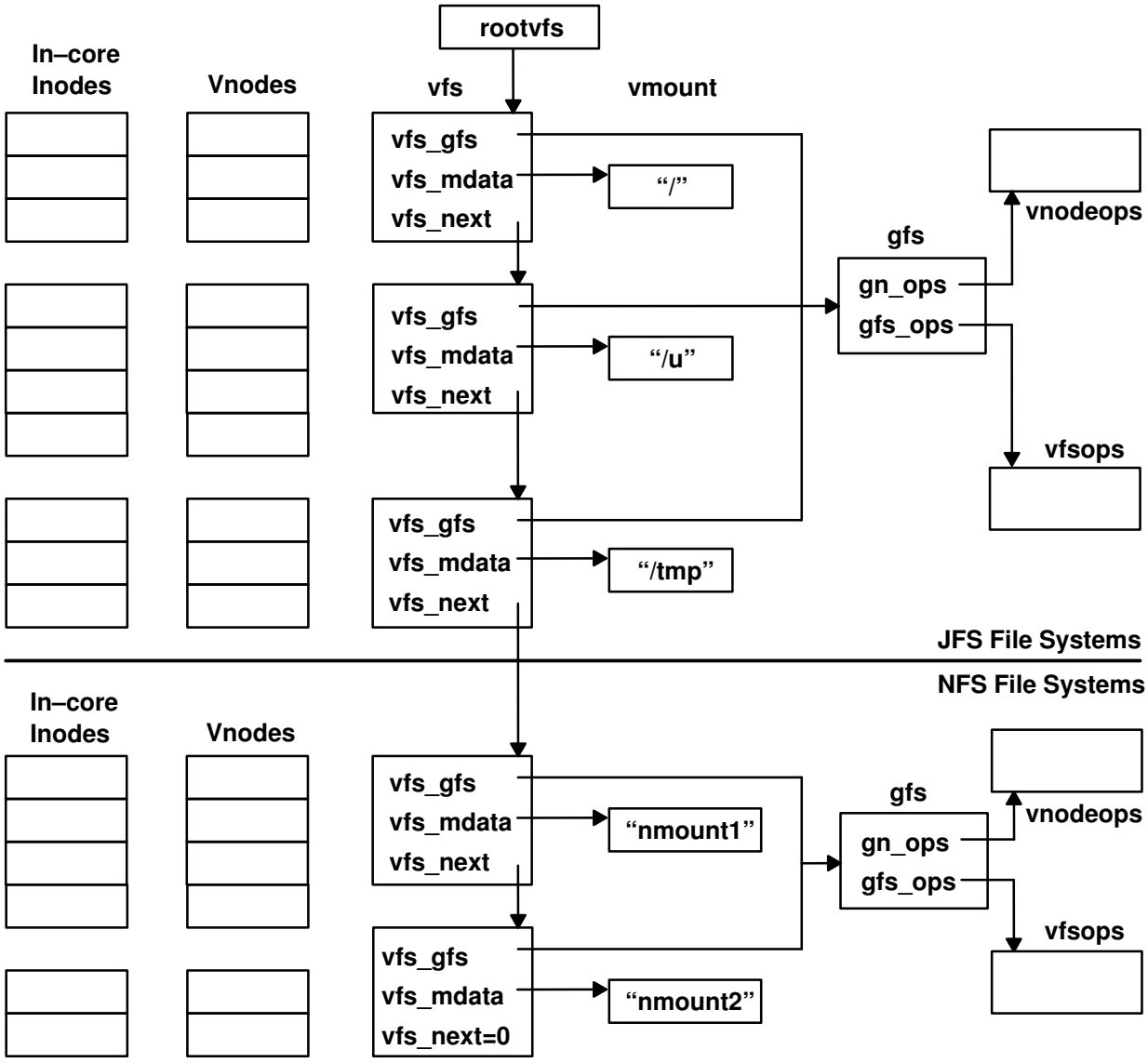
Chapter 10. Writing a Virtual File System

In addition to Journaled File System (JFS), Network File System (NFS), and the CD-ROM file system types included within AIX, it is possible to write your own Virtual File System (VFS). This may be desirable if you want to incorporate a new concept such as a distributed file system, or to use a new device such as a WORM drive or tape jukebox, or simply for performance reasons where a large amount of a specific type of data needs to be managed in an unusual manner.

Multiple File System Types within the Kernel

Each type of file system is represented within the kernel by a *struct gfs*. Each mounted file system is represented by a *struct vfs* which contains a pointer to the appropriate *struct gfs*. These structures are the basis for file system related operations.

The **vfs** structures are in a linked list, regardless of file system type, with the kernel variable *rootvfs* pointing to the first **vfs** structure in the list. The following vfs and gfs Structures figure shows the relationship between **vfs** and **gfs** structures.



vfs and gfs Structures

The **gfs** structures each contain a pointer to a *struct vnodeops*, and a *struct vfsops*. These structures contain a list of functions which have a standardized interface by which system calls can invoke file related (with vnodeops), and file system related (with vfsops) operations.

You add new file system types to the kernel by loading the kernel extension into the kernel using the **sysconfig(KLOAD,...)** subroutine and then invoking the config entry point of the new kernel extension. The kernel extension contains the virtual file system dependent functions.

This in turn creates a **vnodeops** structure and a **vfsops** structure, and initializes these structures with addresses of the functions within the extension. A temporary **gfs** structure is created, and the **gfsadd** kernel service is used to insert the gfs record into the kernel's array of **gfs** structures. At this point, the virtual file system type is usable, and file systems of the new type can be mounted.

Data Structures within a Virtual File System

As already described, a **gfs**, **vnodeops**, and **vfsops** structure are created each time a new file system kernel extension is added to the kernel.

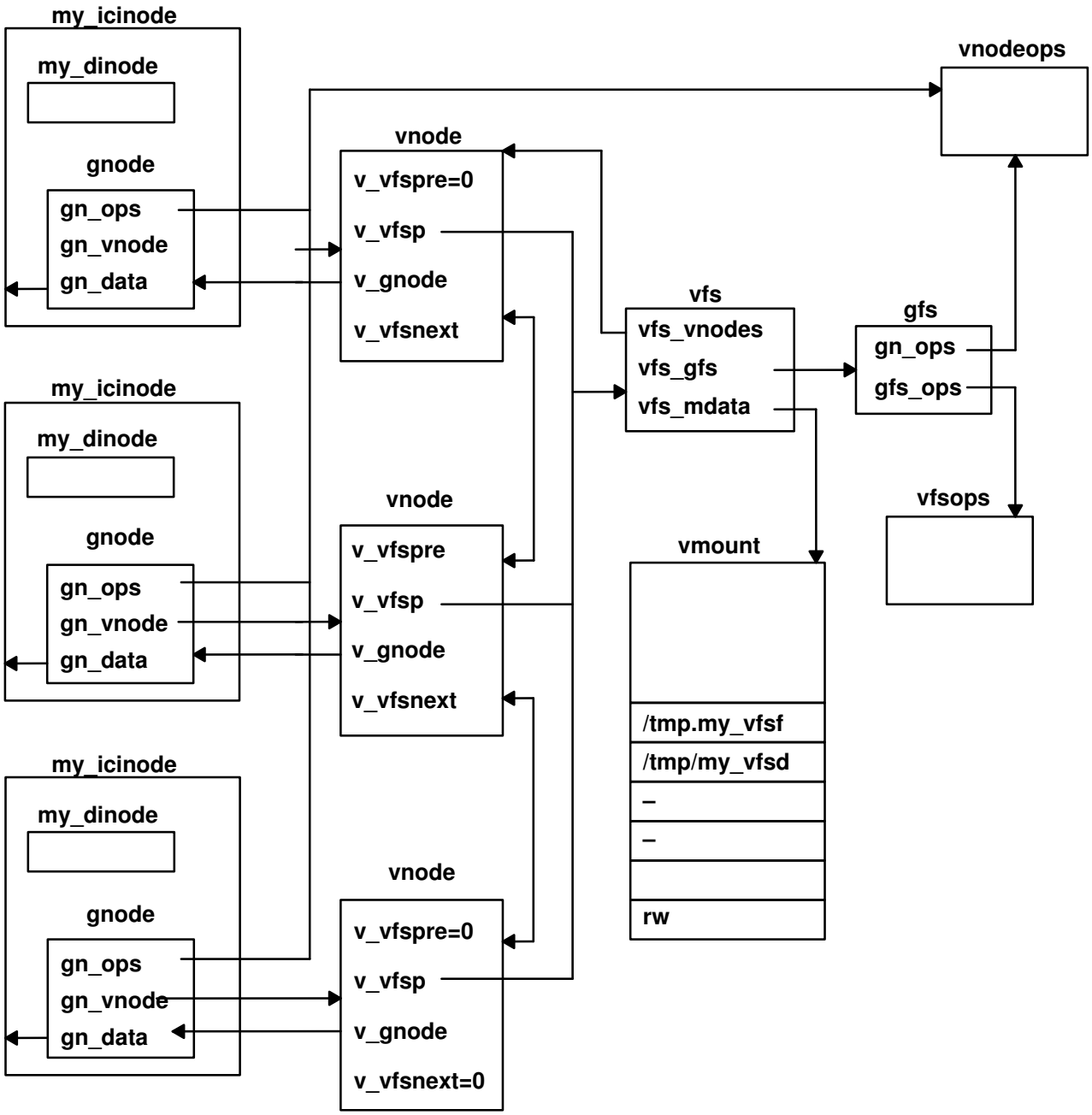
When new file systems are mounted, a **vfs** and **vmount** structure are created. The **vmount** structure contains specifics of the mount request, such as the object being mounted, and the stub over which it is being mounted. The **vfs** structure is the connecting structure which links the vnodes (representing files) with the vmount information, and the **gfs** structure.

The mount helper creates the **vmount** structure, and calls the **vmount** subroutine. The **vmount** subroutine then creates the **vfs** structure, partially populates it, and invokes the file system dependent **vfs_mount** subroutine which completes the **vfs** structure, and performs any operations required internally by the particular file system implementation. See "Mount Helper" on page 10-17 for more information about the mount helper.

Whenever a file is accessed, it is represented by a **vnode** structure, and an in-core **inode** structure, which also contains a **gnode** structure, and possibly a copy of the disk i-node. The **vnode** and **gnode** structures are file system independent, and the kernel can access them directly. However, the kernel has no information about the internal storage of the data files that the vnodes represent. To perform actions on a data file, the kernel passes a vnode pointer to the relevant vnode operation listed in the **vnodeops** structure.

The following File System Data Structures figure shows the relationships of the various data structures within a mounted sample file system `my_fs`. Note that in this file system type, there is a copy of the disk i-node (`my_dinode`) within each in-core i-node (`my_icinode`).

This figure also shows the way that the in-core i-node contains the gnode, which in turn contains pointers both to the containing in-core i-node (gn_data), and a direct link back to the **vnodeops** structure (gn_ops) for efficiency.



File System Data Structures

The following sections contain descriptions of each of the file-system-independent structures.

gfs Structure

The **gfs** structure is defined in the **sys/gfs.h** header file:

```
struct gfs {
   struct vfsops            *gfs_ops;
   struct vnodeops          *gn_ops;
   int  gfs_type;           /* type of gfs (from vmount.h) */
   char     gfs_name[16];   /* name of vfs (eg. "jfs","nfs", .)*/
   int  (*gfs_init)();      /* ( gfsp ) - if ! NULL, */
                            /* called once to init gfs */
   int gfs_flags;           /* flags for gfs capabilities */
   caddr_t  gfs_data;       /* ptr to gfs's private config data*/
   int      (*gfs_rinit)();
   int      gfs_hold        /* count of mounts of the ...  */
}
```

There is one **gfs** structure for each type of virtual file system currently installed on the machine. For each gfs entry, there may be any number of vfs entries.

The **gfs** structures are stored within a global array accessible only by the kernel. The gfs entries are inserted with the **gfsadd** kernel service. Only one gfs entry with a given gfs_type can be inserted into the array. Generally, gfs entries are added by the CFG_INIT section of the configuration code of the file system kernel extension.

The gfs entries are removed with the **gfsdel** kernel service. This is usually done within the CFG_TERM section of the configuration code of the file system kernel extension.

The operating system uses the gfs entries as an access point to the virtual file system functions on a type-by-type basis. There is no direct link from a gfs entry to all of the vfs entries of a particular gfs type. The file system code generally uses the **gfs** structure as a pointer to the **vnodeops** structure and the **vfsops** structure for a particular type of file system, although the gnodes also contain a pointer to the **vnodeops** structure for their type of file system.

vfs structure

The **vfs** structure is defined in the **sys/vfs.h** header file:

```
struct vfs {
   struct vfs    *vfs_next;      /* vfs's are a linked list */
   struct gfs    *vfs_gfs;       /* ptr to gfs of vfs */
   struct vnode  *vfs_mntd;      /* pointer to mounted vnode, */
                                 /*      the root of this vfs */
   struct vnode   *vfs_mntdover; /* pointer to mounted-over */
                                 /*      vnode */
   struct vnode   *vfs_vnodes;   /* all vnodes in this vfs */
   int            vfs_count;     /* number of users of this vfs
*/
   caddr_t        vfs_data;      /* private data area pointer */
   unsigned int   vfs_number;    /* serial number to help */
                                 /*  distinguish between */
                                 /*  different mounts of the */
                                 /*  same object */
   int            vfs_bsize;     /* native block size */
   short          vfs_rsvd1;     /* Reserved */
   unsigned short vfs_rsvd2;     /* Reserved */
   struct vmount  *vfs_mdata;    /* record of mount arguments */
};
```

There is one **vfs** structure for each file system currently mounted.

New **vfs** structures are created with the **vmount** subroutine. This subroutine calls the **vfs_mount** subroutine found within the **vfsops** structure for the particular virtual file system type.

The vfs entries are removed with the **uvmount** subroutine. This subroutine calls the **vfs_umount** subroutine from the **vfsops** structure for the virtual file system type.

The **vfs** structure is central to each mounted file system. It provides access to the vnodes currently loaded for the file system, mount information through the *vfs_mdata* pointer, and provides a path back to the **gfs** structure and its file system specific subroutines through the *vfs_gfs* pointer.

The **vfs** structures are a linked list with the first vfs entry addressed by the *rootvfs* variable which is private to the kernel. The *vfs_mntd* pointer points to the vnode within the file system which generally represents the root directory of the file system. The *vfs_mntdover* pointer points to a vnode within another file system, also usually representing a directory, which indicates where the file system is mounted. In this sense, the *vfs_mntd* pointer corresponds to the object within the **vmount** structure referenced by the *vfs_mdata* pointer, and the *vfs_mntdover* pointer corresponds to the stub within the **vmount** structure referenced by the *vfs_mdata* pointer.

Refer to the “Mount Helper” section on page 10-17 for details of the vmount structure.

vnode structure

The **vnode** structure is defined in the **sys**/**vnode.h** header file:

```
struct vnode {
        ushort  v_flag;         /* see definitions below                */
        ulong   v_count;        /* the use count of this vnode          */
        int     v_vfsgen;       /* generation number for the vfs        */
        Simple_lock v_lock;     /* lock on the structure                */
        struct vfs *v_vfsp;     /* pointer to the vfs of this vnode     */
        struct vfs *v_mvfsp;    /* pointer to vfs which was mounted over this */
                                /* vnode; NULL if no mount has occurred */
        struct gnode *v_gnode;  /* ptr to implementation gnode          */
        struct vnode *v_next;   /* ptr to other vnodes that share same gnode */
        struct vnode *v_vfsnext; /* ptr to next vnode on list off of vfs */
        struct vnode *v_vfsprev; /* ptr to prev vnode on list off of vfs */
        union v_data {
                void *          _v_socket;       /* vnode associated data */
                struct vnode *  _v_pfsvnode;     /* vnode in pfs for spec */
        } _v_data;
        char *  v_audit;         /* ptr to audit object                  */
};
```

Vnodes are the primary handles by which the operating system references files. Most vnode operations are passed a pointer to a vnode as their first parameter.

Vnodes are created by the vfs-specific code when needed, using the **vn_get** kernel service.

Vnodes are deleted with the **vn_free** kernel service.

Vnodes are the result of a path resolution. Each time an object (file) within a file system is located (even if it is not opened), a vnode for that object is located (if already in existence), or created. Naturally, vnodes for each directory searched to resolve the path are also created, or referenced. Vnodes are also created for files as the files are created.

The **vnode** structure provides the link between the **vfs** structure and the **gnode** structure. There are two vnodes for one gnode only in the case of file-over-file mounts. In this case, the gnode refers to the first related vnode, and the other vnodes for that gnode are linked using the `v_next` field.

gnode structure

The **gnode** structure is defined in the **sys/vnode.h** header file:

```
struct gnode {
        enum vtype gn_type;             /* type of object: VDIR,VREG,... */
        short   gn_flags;               /* attributes of object          */
        ulong   gn_seg;                 /* segment into which file is mapped */
        long    gn_mwrcnt;              /* count of map for write        */
        long    gn_mrdcnt;              /* count of map for read         */
        long    gn_rdcnt;               /* total opens for read          */
        long    gn_wrcnt;               /* total opens for write         */
        long    gn_excnt;               /* total opens for exec          */
        long    gn_rshcnt;              /* total opens for read share    */
        struct vnodeops *gn_ops;
        struct vnode *gn_vnode; /* ptr to list of vnodes per this gnode */
        dev_t   gn_rdev;        /* for devices, their "dev_t" */
        chan_t  gn_chan;        /* for devices, their "chan", minor's minor */

        Simple_lock     gn_reclk_lock;  /* lock for filocks list         */
        int             gn_reclk_event; /* event list for file locking   */
        struct filock  *gn_filocks;     /* locked region list            */

        caddr_t gn_data;        /* ptr to private data (usually contiguous) */
};
```

A gnode refers directly to a file (regular, directory, special, and so on), and is usually embedded within a file system implementation specific in-core i-node.

Gnodes are created as needed by file system specific code at the same time as creating in-core i-nodes. This is normally immediately followed by a call to the **vn_get** kernel service to create a matching vnode.

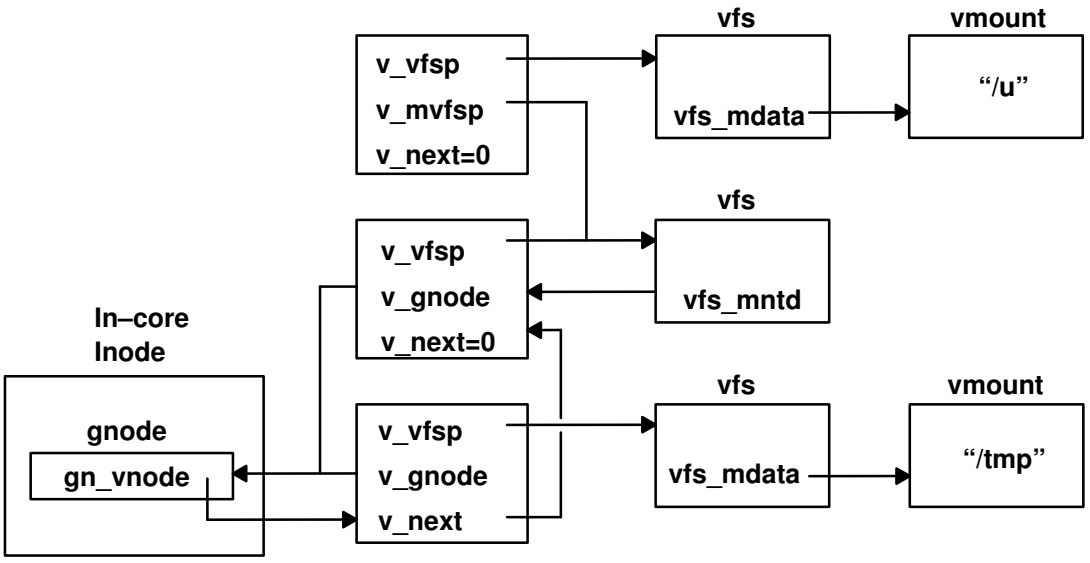
The gnode structure is usually deleted either when the file it refers to is deleted, or when its in-core i-node is removed to make room for an in-core i-node representing a more recently accessed file.

In early UNIX virtual file system implementations, the vnode represented the file system independent information about a file, and the in-core i-node was totally file system implementation specific. In AIX, the gnode is a part of the in-core i-node which is uniform, but the remainder of the in-core i-node remains implementation specific.

File-Over-File Mounts

It is possible to have more than one vnode referring to a gnode. This occurs when a file is mounted over another file. In this case, the `v_mvfs` field within the vnode of the file which is being mounted over is set to point to a newly created **vfs** structure. The new **vfs** structure represents a file system with one file, whose vnode points to the gnode of the file which is mounted.

The following File-Over-File Mounts figure shows the situation where a file in /**tmp** is mounted over a file in /**u**. Note that the gnode points to one of the vnodes which is linked to the next vnode for that gnode with the `v_next` field.



File-Over-File Mounts

Components of a Third-Party Virtual File System

The following are the basic components that you must provide to implement a virtual file system for the AIX operating system:

1. The kernel extension containing initialization functions, vfs operations, vnode operations, and virtual memory operations
2. A file system helper or some other mechanism to create file systems on media
3. A mount helper or a specialized mount command
4. A configuration program to load the kernel extension into the kernel
5. A software installation package (preferably an installp package) to install the software on a customer's computer, and modify /**etc**/**vfs**

Creating the Virtual File System Kernel Extension

The virtual file system kernel extension is similar to a driver in that it runs in kernel mode and has a configuration entry point. Considerations for compiling and linking are also similar.

The following is an example of Makefile entry for building a virtual file system kernel extension:

```
CFLAGS  =  -O -D_AIX -D_SUN -D_KERNEL  -DKERNEL -I../.. -I. -qxref -qlist
CDEFS   = -U_STR_ -U_MATH_ -D_KERNEL -D_AIX -D_IBMR2    \
          -DMACHINE=_IBMR2 '-DMACHNAME="R2_System"'      \
          -DNLS -D_NLS -DMSG -Daiws -Dunix

all:    my_vfs

my_vfs: my_vfsops.o my_vnopeops.o
          /bin/ld -b"binder:/usr/lib/bind glink:/lib/glink.o" \
             -s -D0 -H512 -T512                               \
             -lcsys -lsys                                     \
             -b"I:/lib/kernex.exp I:/lib/syscalls.exp E:myfs_t.exp
map:sym.myfs_t"                          \
             -e myfs_config              \
             my_vfsops.o my_vnodeops.o   \
             -o my_vfs

my_vnodeops.o:  my_vnodeops.c myfs.h
          ${CC} ${CFLAGS} ${CDEFS} -c my_vnodeops.c 2> $*.err

my_vfsops.o:    my_vfsops.c myfs.h
          ${CC} ${CFLAGS} ${CDEFS} -c my_vfsops.c 2> $*.err
```

Note: To bring in the appropriate version of printf and have debugging statements appear on the terminal, do *not* use **cc** for the linking phase of the operation.

Entry Points within the File System Kernel Extension

This section describes the **config**, **init**, and **rootinit** entry points.

config

```
int myfs_config (int cmd, struct uio *uiop);
```

The primary entry point within the file system kernel extension is the config entry point. The configuration program calls this entry point once the kernel extension is loaded. The configuration routine to which the config entry point refers usually consists of a case statement which switches based on the command passed in. The two cases which should be taken into account are **CFG_INIT** and **CFG_TERM**. However, you can add more commands.

The **CFG_INIT** branch should:

1. Check that the extension has not already been initialized.
2. Initialize any private data structures.
3. Create and add an entry into the gfs table using the **gfsadd** kernel service. This calls the init entry point from the **gfs** structure.
4. Register the page fault handler (**strategy** routine) using the **vm_mount** kernel service.
5. Pin any code or data within the file system which must not be paged. For example, pin interrupt handlers or the strategy routine.

The **CFG_TERM** branch should:

1. Verify that it is safe (appropriate) to terminate the extension. For example, verify that the file system has been unmounted.
2. Remove the **gfs** struct entry using the **gfsdel** kernel service.
3. Unpin the areas pinned during the **CFG_INIT**.

init

```
int myfs_init (struct gfs *gfsp);
```

This routine is called by the **gfsadd** kernel service immediately after creating the **gfs** structure. The address of the function is passed into **gfsadd** within the `gfs_init` field of the **gfs** structure. Any functions that must be performed after the gfs entry is created but before users can start to use the file system should be performed here.

The most common operation within this function is to initialize file system dependent structures. The function may also spawn a pager process using the **creatp** and **initp** kernel services.

rootinit

This function is only called for the root file system. The entry point is passed to the **gfsadd** kernel service in the `gfs_rinit` field of the **gfs** structure. The most common operations are to initialize the root **vmount** and **vfs** structures, and to open the root device.

VFS Operations within the File System Kernel Extension

The vfs operations are those operations which affect an entire file system, such as:

- **mount**
- **unmount**
- **sync**

The addresses of these functions are stored within a **vfsops** structure that is pointed to by the `gfs_ops` field of the **gfs** structure for a particular virtual file system implementation.

For specific details of the vfs operations requirements, see the *AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts* book.

The following are descriptions of the vfs operations for a typical virtual file system.

vfs_mount

```
int myfs_mount(struct vfs *vfsp, struct ucred *crp);
```

This routine is called by the **vmount** subroutine, and performs the following actions in addition to any internal work required:

1. Extract important data passed by the mount helper program in the *vfsp–>vfs_mdata* field.
2. Obtain the vnode of the object being mounted with the **lookupvp** kernel service.
3. Check that the object being mounted is appropriate.
4. Initialize the following fields within the **vmount** structure (pointed to by *vfsp–>vfs_mdata*):
   - `vmt_fsid`
   - `vmt_vfsnumber`
   - `vmt_time`
5. If applicable, call vm_mount to register the file system's strategy routine.

6. Create a vnode (and matching in-core i-node) for the root directory of the file system, and initialize the `v_mvfsp` field of the stub vnode (pointed to by *vfsp–>vfs_mntdover*) with its address.
7. Use the **vn_rele** vnode operation to release the object's vnode.

vfs_unmount

```
int myfs_unmount(struct vfs *vfsp,int flags, struct ucred *crp);
```

This routine is called by the **umount** subroutine. This routine performs the following actions:

1. Free up any resources used by this mount, including vnodes, using the **vn_free** vnode operation.
2. Set the `v_mvfsp` field of the vnode for the stub to NULL.
3. Set *vfsp–>vfs_mntd* to NULL.
4. Call the **vm_umount** kernel service to unregister the file system's strategy routine.
5. Call the **vfsrele** kernel service to release the system-created resources.

vfs_root

```
int myfs_root(struct vfs *vfsp, struct vnode **vpp,
               struct ucred *crp);
```

This routine is frequently called when locating files within a file system. Essentially, it finds, or creates a vnode (associated with an in-core i-node) for the root of the file system (that is, the file containing the root directory). Before returning success, the routine should check that the file system is mounted, and calls the **vn_hold** vnode operation for the vnode it is about to return.

vfs_statfs

```
int myfs_statfs(struct vfs *vfsp, struct statfs *statfsp,
                 struct ucred *crp);
```

The **vfs_statfs** routine returns basic file system statistics in the form of a **statfs** structure. See the **sys/statfs.h** header file for the fields of the **statfs** structure. The methods to obtain this information are implementation dependent.

vfs_sync

```
int myfs_sync();
```

Every 60 seconds, and whenever the **sync** command is run, the **sync** function is called. This in turn calls the **vfs_sync** function *once* for each different file system type. Note that a pointer to a **gfs** structure is passed in.

The actual use of this routine is totally implementation dependent, but once completed each file system of the same **gfs** type should be updated on secondary storage (assuming there is any) to the point where it is consistent.

vfs_vget

See "File System Operations" in *AIX Version 4.1 Technical Reference, Volume 5: Kernel and Subsystems*.

vfs_cntl

See "File Systems Operations" in *AIX Version 4.1 Technical Reference, Volume 5: Kernel and Subsystems.*

Vnode Operations within the File System Kernel Extension

The vnode operations are the operations that work on specific files (represented by vnodes) within a file system. This section covers the required vnode operations.

See “File System Operations” in *AIX Version 4.1 Technical Reference, Volume 5: Kernel and Subsystems* for details on the requirements for the vnode operations.

vn_hold

```
int myfs_hold (struct vnode *vp);
```

This routine simply increments the `v_count` field (the usage count) of the vnode.

vn_rele

```
int myfs_rele (struct vnode *vp);
```

This routine decrements the usage count of the vnode. If the usage count is reduced to zero, remove the vnode from memory using the **vn_free** vnode operation.

It is practical to release the related in-code i-node at this point, assuming (as is normal) that this is the only vnode referencing the in-core i-node.

Check that the file system was being unmounted but could not complete because there were still open files (*vfsp–>vfs_flag* & VFS_UNMOUNTING). If this was the last vnode for the file system (*vfsp–>count* == NULL), call the **vfsrele** kernel service to release the vfs entry and complete the unmount.

vn_getattr

```
int myfs_getattr (struct vnode *vp, struct vattr *ubuf,
                  struct ucred *crp);
```

This routine obtains the attributes of the file referenced by *vp* from the related gnode and in-core i-node which is in a file-system specific format, and returns the information in a **vattr** structure.

The following shows the **vattr** structure as defined in the **sys/vattr.h** header file:

```
struct vattr {
   enum vtype va_type;     /* from gnode.gn_type*/
   mode_t     va_mode;     /* access modes from in-core i-node*/
   uid_t          va_uid;  /* user id from in-core i-node*/
   gid_t          va_gid;  /* group id from in-core i-node*/
   union {
      dev_t       _va_dev;
      long        _va_fsid;   /* vfs_mdata->vmt_fsid.fsid_dev*/
   } _vattr_union;
   long   va_serialno;     /* i-node numberfrom in-core i-node*/
   short  va_nlink;        /* number of links to fil from in-core
                              i-node*/
   long   va_size;  /* file size in bytes from in-core i-node*/
   long   va_blocksize;    /* block size for file system */
   long   va_blocks;       /* blocks reserved for file
                              from in-core i-node*/
   struct timeval va_atime;/* last file access time
                              from in-core i-node*/
   struct timeval va_mtime;/* last file modification
                              time from in-core i-node*/
   struct timeval va_ctime;/* last disk i-node modification
                              time from in-core i-node*/
   dev_t          va_rdev; /* from gnode.gn_rdev*/
   long           va_nid;  /* use unamex(), field nid*/
   chan_t     va_chan;  /* from gnode.gn_chan*/
   char       *va_acl;  /* Access Control List*/
   int        va_aclsiz;/* size of ACL*/
   int        va_gen:   /* inode generation number */
};
```

vn_open

```
int myfs_open (struct vnode *vp, int flags, int ext,
                caddr_t vinfop, struct ucred *crp);
```

The **vn_open** routine should check the validity of the operation, in particular any file-system dependent tests, such as the requested open mode conflicting with the mode in which the file is already opened by another process.

The file can be bound in at this point, or the binding can be delayed until the first read or write to the file. See the “Virtual Memory Operations” section on page 10-15 for information on binding.

This routine should also increment the appropriate usage counter within the gnode (*gn_rdcnt*, *gn_wrcnt*, or *gn_excnt*).

vn_close

```
int myfs_close (struct vnode *vp, int flags,caddr_t vinfo,
                  struct ucred *crp);
```

The **close** routine calls **vn_close** routine. Any usage counters incremented by the **vn_open** call should be decremented. Do *not* remove the vnode within **vn_close**, this is done within **vn_rele**.

vn_strategy

```
int myfs_strategy (struct buf *bp);
```

The **strategy** routine is called by the virtual memory manager when a page of memory is referenced that is mapped to a file, but the memory has not been paged in. Usually, this routine simply places the buffer on the list of pending work for the pager, and wakes the pager.

The buffer pointer passed in will have the following values in its fields:

b_bcount Always PAGESIZE (4K).

b_blkno The 512-byte block number of the offset within the file (0, 8, 16, and so on).

b_baddr The 4K offset within the mapped segment. This rolls over to zero once a 256-MB segment is crossed.

av_forw Non-null if there are multiple requests to process. This will happen if the virtual memory manager attempts to perform read ahead operations.

b_flags Set to B_READ for a read operation, B_WRITE for a write operation and (B_READ | B_PFSTORE) for a read before a write operation.

vn_rdwr

```
int myfs_rdwr (struct vnode *vp, enum uio_rw op, int flags,
               struct uio *uiop, int ext, caddr_t vinfo,
               struct vattr *vattrp, struct ucred *crp);
```

This routine performs file (not directory) reads or writes. Follow these procedures:

1. Verify that the file type is correct.
2. Return success if the transfer size is zero.
3. Return EINVAL if the start offset is negative.
4. Verify that the file is mapped. If it is not mapped in, map it using the **vms_create** kernel service.
5. Use **vm_move** to copy the data from the source to the destination, reducing the transfer length on reads that go past the end of the file.

Note: The **vm_move** amounts to a copy which can page fault.

vn_lookup

```
int myfs_lookup(struct vnode *dp, struct vnode **vpp,
                char *name, int flags, struct ucred *crp);
```

This routine finds the file *name* which is the basename of a file in the directory *dp*, and returns a vnode pointer for the file. If the file is not found, set *vpp* to NULL.

vn_readdir

```
int myfs_readdir (struct vnode *vp, struct uio *uiop,
                  struct ucred *crp);
```

This routine is called to read directories, which it returns in file-system independent format using a **dirent** structure. Even **read** and **fread** call this routine if the file is a directory, and the directory contents are reconstituted (imperfectly) from the structures.

Read directory entries from the directory starting with the first entry at or beyond *uiop–>uio_offset*. Entries are read into an internal buffer while the buffer size is less than the *uio–>resid* passed in (which is 4K), or until the end of the directory is reached. You do not need to place empty directory entries in the buffer. The buffer should be filled in with as many entries as possible.

The buffer is then copied back into user space using the **uiomove** kernel service which updates *uio–>uio_resid*. The **vn_readdir** routine then updates *uiop–>uio_offset* to be the offset within the directory of the next unread entry.

If the routine is called with no directory entries beyond the *uio_offset*, the routine should return with *uio_resid* untouched.

The following listing shows the fields of the **dirent** structure as found in the **sys/dir.h** header file:

```
#define _D_NAME_MAX 255
struct  dirent {
   ulong_t        d_offset;   /* offset within directory file
                                of next directory entry */
   ino_t      d_ino;          /* i-node number of entry */
   ushort_t   d_reclen;       /* Use DIRSIZ(struct dirent *)
                                after d_namlen is initialized */
   ushort_t   d_namlen;       /* strlen(d_name field)*/
   char       d_name[_D_NAME_MAX+1];
      /* name must be no longer than this including the '\0' */
};
```

Virtual Memory Operations

Once a file is opened, it is bound into an address range. This address range is not in either the user or kernel space, but is inserted into the kernel address space on a temporary basis by **vms_create**.

Accesses to this range may cause page faults which cause the registered **strategy** routine to be called. The **strategy** routine then arranges for the information to be paged in or out without itself page faulting. For this reason, pin the **strategy** routine and any data areas it uses.

A separate process can be started to handle page faults as requested by the **strategy** routine, and this process is under no obligation to avoid page faulting.

Binding the File to an Address Range

To bind a file to an address range, call the **vms_create** kernel service. This reserves part of the 52-bit virtual memory address space of the machine, and associates it with the gnode passed in. The vmid passed back is effectively a segment register value, which should be placed in the `gn_seg` field of the **gnode** structure. This is typically done by the file system's open routine or by the read/write function (upon the first I/O attempt).

On subsequent reads or writes to the file, the **vn_rdwr** subroutine passes this vmid to **vm_move**, along with transfer length, uio pointer and so on. Then **vm_move** places the address space of the file within the kernel address space long enough to perform the transfer.

For more information on **vms_create**, refer to kernel services in *AIX Version 4.1 Technical Reference, Volume 5: Kernel and Subsystems.*

The Page Handling Process

The pager for a virtual file system may run as a separate process, a process spawned by the **vfs_init** subroutine. The routine uses the **e_sleep_thread** kernel service on a list of buf structures to which the **vn_strategy** subroutine appends further entries. Once the list contains at least one request, the pager parses the **buf** structures, and in many virtual file system implementations, attempts to order them depending upon the situation.

For instance, buffers may be separated into requests for different media so that one disk does not need to wait until another disk has completed a series of transfers. Or the requests may be ordered to optimize disk seek sequencing.

Paging a Block In or Out

The actual technique used to page a block in or out varies greatly between virtual file systems. In a situation where the file system resides directly on media, such as in the CD-ROM file system, the **buf** structures are modified to contain the actual device block number, and device IDs, then the **devstrat(bufp)** kernel service is called by the page fault handler. This calls the **strategy** routine within the device driver to perform the transfer.

Once the data is transferred, call the **iodone(bp)** kernel service to notify the virtual memory manager that the memory space is now valid.

File System Helper

The file system helper for a virtual file system type is listed in the /**etc**/**vfs** file. This program is called by AIX system management programs, usually to operate on an unmounted file system. Examples include:

fsck Calls the file system helper to perform an implementation-dependent check of the contents of a file system.

mkfs Invokes the file system helper to create a file system.

chfs Uses the file system helper to extend a file system.

An example of an entry in the /**etc**/**filesystems** file follows:

```
/mymnt:
        dev             = /dev/fd0
        vfs             = myfs
        mount           = true
        options         = rw
        type            = myfs
        nodename        = myfs
```

For the preceding example, the command:

```
mkfs /mymnt
```

translates to a call to the file system helper with the following parameters:

```
fshop_make 5 5 7 -ip 0 \
 name=/tmp/my_vfsd,label=/tmp/my_vfsd,dev=/tmp/my_vfsf
```

Note: The `fshop_make` is *not* the name of the file system helper. The subcommand **argv[0]** is set to a name other than the program name using the **execlp** subroutine.

We suggest that third-party virtual file system writers use **argv[1]**, which is a command from the **fshelp.h** file such as **FSHOP_MAKE**.

The command:

```
fsck -v myfs
```

translates to the following call:

```
fshop_check 1 3 6 -ifp 0 device=/dev/hd1,mounted
```

The basic subcommands are:

argv[A_NAME (=0)]
The command in `char *` format

argv[A_OP (=1)]
The command as an entry from the **fshelp.h** file

argv[A_FSFD (=2)]
File descriptor number for the file system

argv[A_COMFD (=3)]
File descriptor number for pipe

argv[A_MODE (=4)]
Mode flags. For example, i = **FSHMOD_INTERACT_FLAG**

argv[A_DEBG (=5)]
Debug level

argv[A_FLGS (=6)]
Operation-dependent flags

The functions performed by any file system helper are not fixed. Implement only those appropriate to a virtual file system. Furthermore, it is acceptable for some functions to be omitted, and implemented as separate programs, although the AIX commands will not be able to invoke them.

Mount Helper

The format of the mount helper is similar to that of the file system helper. The mounting program may be in any format, but for it to be invoked by the AIX **mount** command it should be listed as the mount helper in the /**etc**/**vfs** file, and conform to the following invocation format:

argv[0] The command path

argv[1] Command (M=mount, U=unmount)

argv[2] Debug Level

argv[3] Nodename

argv[4] Object (the file system being mounted)

argv[5] Stub (the directory that the file system is mounted over)

argv[6] Options (for example, “rw” = read/write)

The basic operations of the mount helper are:

1. Checks that the operation is valid.
2. Allocates space for a **vmount** structure with enough space at the end for the additional parameters. See the **sys**/**vmount.h** header file for information on the **vmount** structure.
3. Populates the **vmount** structure.
4. Calls **vmount (vmountptr, size_of_vmount_struct)**.

The contents of the **vmount** structure are defined as follows:

```
#define VMT_OBJECT   0   /* I index of object name */
#define VMT_STUB     1   /* I index of mounted over stub name */
#define VMT_HOST     2   /* I index of (short) hostname */
#define VMT_HOSTNAME 3   /* I index of (long) hostname */
#define VMT_INFO     4   /* I index of binary vfs specific info
*/
                         /* includes network address, opts, etc*/
#define VMT_ARGS     5   /* I index of text of vfs specific args*
/
#define VMT_LASTINDEX 5  /* I the last in the array of structs */
struct vmount {
   ulong   vmt_revision; /* revision level, currently 1 */
   ulong   vmt_length;   /* length of structure and data */
   fsid_t  vmt_fsid;     /* Do not set */
   int     vmt_vfsnumber; /* Do not set */
   time_t  vmt_time;     /* Do not set */
   ulong   vmt_timepad;  /* Do not set */
   int     vmt_flags;    /* general mount flags */
   int     vmt_gfstype;  /* type of gfs, e.g. MNT_JFS */
   struct  vmt_data  {
      short vmt_off;     /* I offset of data, word aligned */
      short vmt_size;    /* I actual size of data in bytes */
   }  vmt_data[VMT_LASTINDEX + 1];
};
```

Note: The `vmt_off` field is the offset from the start of the **vmount** structure to the start of the data element.

The following is an example of a hexadecimal dump of a **vmount** structure:

```
0000: 00000001 0000006c 00000000  00000000 .... ...l  .... ....
0010: 00000000 00000000 00000000  00000000 .... ....  .... ....
0020: 00000008 003c000e 004c000e  005c0002 .... .<..  .L.. .\..
0030: 00600002 00640004 00680003  2f746d70 .`.. .d..  .h.. /tmp
0040: 2f64656d 6f766673 66000000  2f746d70 /dem ovfs  f... /tmp
0050: 2f64656d 6f766673 64000000  2d000000 /dem ovfs  d... -...
0060: 2d000000 00000000 72770000  00000000 -... ....  rw.. ....
```

Virtual File System Configuration Program

You must load the virtual file system kernel extension in much the same way as a driver. This operation can be performed by a separate program that is invoked by an rc script, or preferably, as a rule in the Config_Rules database. It can also be invoked by the mount helper when the first file system of a particular type is mounted.

Load the kernel extension with the **sysconfig** subroutine:

```
sysconfig (SYS_KLOAD,...)
```

Then use **sysconfig** to call the configuration entry point of the extension:

```
sysconfig (SYS_CFGKMOD,...)
```

The following is an example of using the **sysconfig** subroutine:

```
struct cfg_load load;
struct cfg_kmod kmod;
.
.
.
load.path = kern_ext_name;
if( sysconfig( SYS_KLOAD, &load, sizeof(load) ) == -1 ){
        fprintf (stderr, "Unable to SYS_KLOAD %s, errno = %d\n",
                    load.path, errno );
        exit(1);
}
        fprintf( stderr, "loaded at 0x%08x\n", load.kmid );

/* initialize extension */
kmod.kmid = load.kmid;
kmod.cmd = CFG_INIT;
kmod.mdiptr = (caddr_t) &kmod.kmid;
kmod.mdilen = sizeof( kmod.kmid );

if( sysconfig( SYS_CFGKMOD, &kmod, sizeof(kmod)) == -1 ){
        fprintf( stderr,
                "Unable to configure module, errno=%d\n",
                errno);
        exit(1);
}
```

During development, and for some file system implementations, you may want to unload the kernel extension. To do this, use the following calls:

```
kmod.cmd = CFG_TERM;
sysconfig (SYS_CFGKMOD, &kmod,...);
sysconfig (SYS_KULOAD, &load,...);
```

Software Installation Package

You should package your software in the form of an installp module, because this is the form in which customers expect to receive software for AIX Version 4.1. See “Software Product Packaging” in *AIX Version 4.1 General Programming Concepts: Writing and Debugging Programs* for more information.

In addition to installing the software in the customer’s system, the installation program also:

1. Appends an entry into the /**etc**/**vfs** file for the new file system type. For example:

```
my_fs    8  /etc/helpers/my_fsmnthelp   /etc/helpers/my_fshelper
```

2. (Optional) Inserts a new rule into the Config_Rules database to invoke the kernel extension installation program during system startup.
3. (Optional) Creates an initial entry in /**etc**/**filesystems**.

/etc/vfs

The /**etc**/**vfs** file describes the currently installed virtual file systems. Entries include:

- Comment lines, which start with a #.
- The general control line which defines the default local, and (optionally) the default remote virtual file system. This is typically:

```
%defaultvfs     jfs     nfs
```

- VFS entries that consist of:

  name — The name of the vfs type.

  type — The number of the vfs type.

  mount-helper — The name of the mount helper, or “none”. This is with respect to /**etc**/**helpers** unless it starts with a / (slash).

  file system-helper — Same conventions as mount-helper.

  Use white space to separate these fields.

For a user file system, use numbers from MNT_USRVFS (=8) to MNT_AIXLAST (=15) for the type. These values are defined in the **sys**/**vmount.h** header file. File system numbers between zero and MNT_USRVFS – 1 (=7) are reserved for assignment by the operating system.

Once the example file system “my_fs” is inserted, the end of /**etc**/**vfs** contains:

```
%defaultvfs   jfs   nfs
#
cdrfs         5     none    none
jfs           3     none    /etc/helpers/v3fshelper
nfs           2             /etc/helpers/nfsmnthelp
                    none    remote
my_fs         8             /etc/helpers/my_fsmnthelp
                            /etc/helpers/my_fshelper
```

Loading the File System Kernel Extension during System Startup

The easiest way to load the kernel extension into the kernel during system startup is to insert a new rule into the Config_Rules database. To do this, create a file with the new rule in it. Our sample file is named **my_fs_rule.add**:

```
Config_Rules:
    phase = 2
    seq = 0
    rule = ”/etc/methods/installmy_fs -a /etc/drivers/my_fs”
```

To install the rule, use the following ODM commands:

```
odmdelete -q "rule LIKE '*my_fs*'" -o Config_Rules
odmadd my_fs_rule.add
```

The **odmdelete** command deletes old Config_Rules entries which may have been inserted for this virtual file system. If you use the **odmadd** command twice without **odmdelete**, there will be two rules in the database, and the extension installation program will be invoked twice.

Do *not* use this technique during development, as a faulty kernel extension could prevent a successful system startup. Instead, add the rule at the end of the development cycle when the extension is stable.

Virtual File System Terminology

Disk i-node A disk i-node is an i-node that resides on a disk (or some other secondary storage). Generally, this i-node contains all of the information about a file which also resides on the disk except its name (which is in one or more directories), and its contents (which are stored in the data area reserved for the file). This information includes file type, size, permissions, access times, and data location information (assuming the file is a data file, not a special file). Disk i-nodes are referred to by their i-node number which is unique within a particular file system.

gfs entry A **gfs** structure exists for each TYPE of virtual file system that is being used within the system. For example, there is one **gfs** structure for the Journaled File System (JFS) regardless of the number of JFS file systems currently mounted, and if there are any NFS mounts, there is one **gfs** structure for NFS. The **gfs** structure provides the system with a set of pointers to the standardized functions for a file system type.

gnode A gnode is a data structure containing information about a file that is currently (or has recently) being referenced. The gnode is effectively an extension of the vnode, and contains the information which is directly associated with the file, independent from the vfs by which the file is being accessed. There can only be one gnode within a system for a particular file at any one time. The gnode is a system-defined structure (that is, vfs independent), contained within an in-core i-node which is vfs dependent.

in-core i-node An in-core i-node is a data structure maintained by a file system kernel extension which represents a file on disk. The structure of the in-core i-node is implementation dependent; however, it contains a gnode structure that is implementation independent. It is common that the in-core i-node also contain a copy of the disk i-node for the file it represents.

virtual file system (**vfs**)
A virtual file system represents a conventional mounted file system. The purpose of having virtual file systems is so that the kernel, and system calls need not know anything about the file system internals, but rather they can perform file system operations, and file operations through a defined interface that is independent of the data storage format or media.

vnode A vnode is a data structure representing a file within a virtual file system. Accesses by system calls to a file are performed using a vnode pointer which provides an implementation independent interface. Each vnode contains a pointer to the vfs which contains it, and to the gnode which contains limited information about the file, and a pointer to the in-core i-node that contains the implementation specific data about the file.

Chapter 11. STREAMS-Based TTY Subsystem Interface

This chapter describes the programming interface of the tty subsystem. This interface can be used by a tty programmer to write a specific module (such as a converter or a line discipline), or a specific driver.

The following information is provided:

- The description of the tty modules and drivers and especially the protocol used in the exchange of STREAMS messages.
- The way the tty subsystem is integrated in a multiprocessor environment.
- The list of the IOCTL operations relating to the tty subsystem.
- The tty data structures extracted from the **str_tty.h** file where tty constants, functions and messages are defined.

The tty subsystem is based on STREAMS. The stack structure of a tty stream is made up of the following modules:

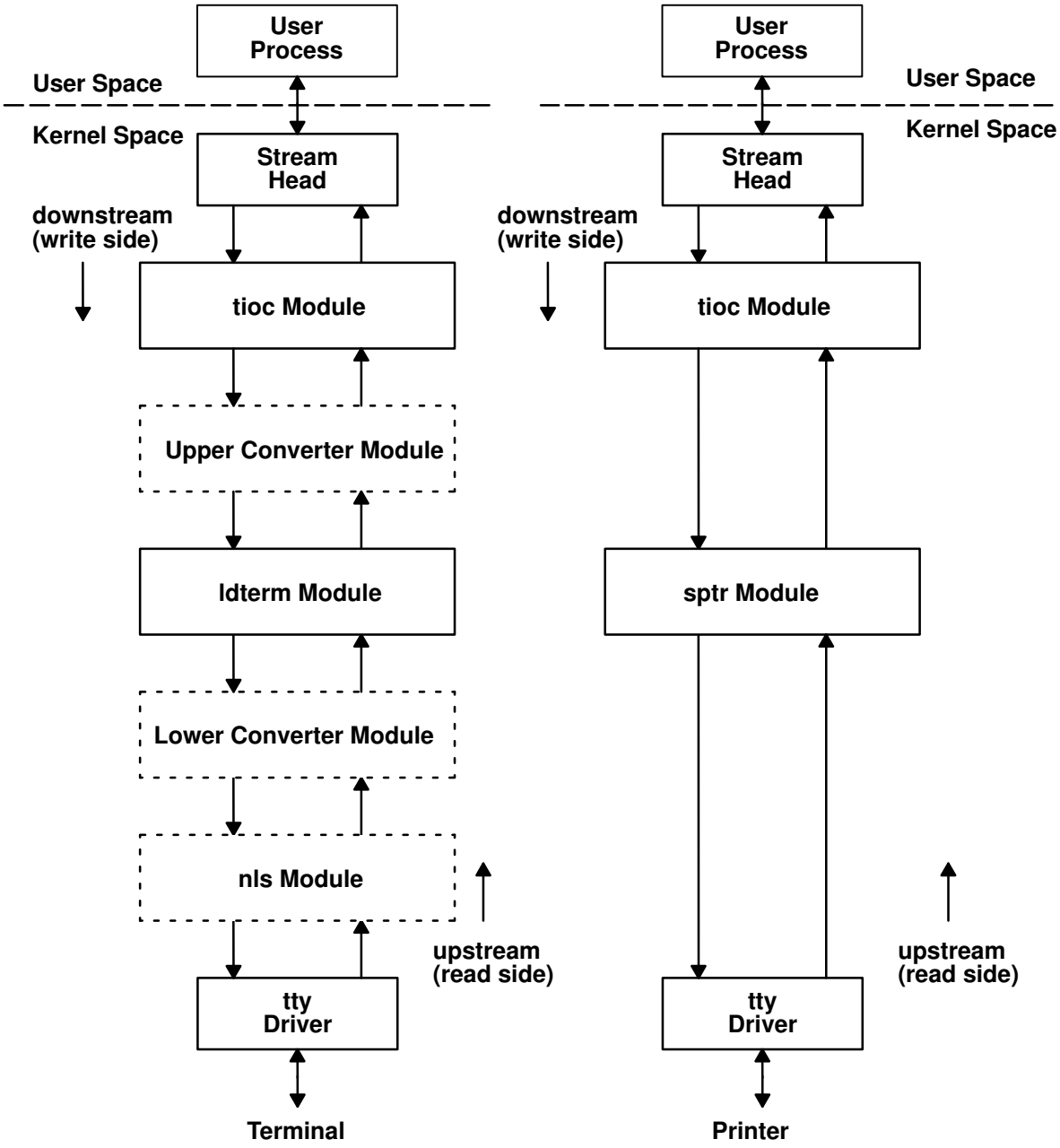
- The stream head, which processes the user's requests. It is common to all tty devices, regardless of the line discipline or tty driver in use.
- The **tioc** module. It is provided to facilitate processing of the transparent ioctl operations.
- The line discipline (**ldterm**, **sptr** or **slip**).
- The stream end. It is a tty driver (**cxma**, **lft**, **lion**, **rs**, or **sf**) or pseudo-driver (**pty**).

The tty stream may also contain:

- An optional character mapping module (**nls**). It is a converter module pushed above the tty driver to support input and output data mapping. The **nls** module uses the input and output map files provided by the **setmaps** command.
- A pair of optional converters to convert upstream and downstream data. An upper converter module is pushed above the line discipline and a lower converter module is pushed below the line discipline. (For example, **uc_sjis** and **lc_sjis** are the upper and lower converters used to convert IBM-932 code set into or from the EUC code set handled by the **ldterm** line discipline.)

Note: The optional modules are not described in this chapter. See "The TTY Subsystem" in *AIX Version 4.1 General Programming Concepts: Writing and Debugging Programs* for more information.

The following figure shows the tty stream structure for a terminal and for a serial printer. The line discipline for a terminal is **ldterm**. The line discipline for a serial printer is **sptr** and the stream structure is simpler (the converters, and **nls** modules are not needed.)



TTY Terminal Stream **TTY Serial Printer Stream**

User modules can also be added or can replace the standard tty modules or drivers to support other specific functionalities.

The means of communication within a stream are messages. Messages are sent through a stream by calls to routines of each queue (write-side or read-side queue) in the stream. Messages can be generated by a driver, a module, or by the stream head. The messages are exchanged between modules or drivers in conformance with protocol rules which are described in the following sections.

Messages that are passed from the stream head toward the driver are said to travel *downstream* (also called *write side*). Similarly, messages passed in the other direction, from the driver to the stream head, travel *upstream* (also called *read side*).

For more general information about tty subsystem and STREAMS, refer to *AIX Version 4.1 General Programming Concepts: Writing and Debugging Programs* and *AIX Version 4.1 Communications Programming Concepts*.

Note: Because the tty subsystem is based on STREAMS, a good knowledge of STREAMS concepts is highly recommended.

Stream Head

The stream head is the interface between the stream and the user application. The stream head processes the user's requests; it is common to all tty devices regardless of the line discipline or tty driver in use.

The stream head performs the following functions:

- It allocates the stream as the controlling terminal if none is already allocated.
- It handles the process group and the session associated with the controlling terminal.
- It handles the job control.
- It processes the **M_PCSIG** and **M_HANGUP** messages coming from the line discipline and generates the appropriate signals to the appropriate user process. It also handles the message type **M_ERROR** from any downstream modules.
- It handles "trusted path" for security purposes. This functionality is shared between the stream head and the driver.
- It handles the "revoke" function which kills all waiting processes attached to the specified file descriptor.

The stream head processes various messages. In particular, the **M_SETOPTS** message is sent by the line discipline module to alter some characteristics of the stream head. The **SO_ISTTY** flag contained in an **M_SETOPTS** message indicates to the stream head that the stream is established for a terminal.

The IOCTLs listed here are directly processed by the stream head:

TIOCSPGRP (or **TXSPGRP**)
Sets the process group identificator for the tty.

TIOCGPGRP (or **TXGPGRP**)
Gets the process group identificator for the tty.

TIOCGSID Gets the session identificator for the tty.

TXISATTY Answers if this stream is a tty or not.

TCTRUST Sets or resets the trust flag in the stream head.

TCQTRUST Gets the state of the tty (trusted or not).

TCSAK Sets or resets the state for secure attention key (SAK).

TCQSAK Identifies the process of a user asking for the SAK sequence.

TIOCCONS Sets the console redirection active or not.

TCXONC Generates the following message types downstream based on the input parameter:

Input Parameter	Message Type
TCOON	**M_START**
TCOOF	**M_STOP**
TCION	**M_STARTI**
TCIOFF	**M_STOPI**

TCFLSH Generates a **M_FLUSH** message type based on input parameter type (**FLUSHR**, **FLUSHW**).

TIOC Module

The IOCTL system calls from applications that are addressed to modules or drivers can be processed according to two STREAMS mechanisms:

- The first mechanism allows the processing of ioctls issued using the **I_STR** call. The associated structure (**strioctl** defined in the **stropts.h** file) contains the IOCTL command and the number of bytes of data. The **I_STR** calls are handled by the stream head and forwarded downstream.
- The second mechanism allows the processing of ioctls other than **I_STR**. These IOCTLs are known as transparent IOCTLs. (The transparent mechanism transparently supports applications developed prior to the introduction of STREAMS.)

If an IOCTL is not processed in the stream head, the stream head creates an **M_IOCTL** message which includes the ioctl arguments, and sends this message downstream for processing by a specific module or driver. **M_IOCTL** messages containing a transparent IOCTL have a TRANSPARENT indicator in their associated **iocblk** structure defined in the **stream.h** file. The **M_IOCTL** messages can be followed by one or more **M_DATA** blocks.

The **tioc** module is provided in the tty subsystem to facilitate processing of the transparent IOCTLs in the lower modules. It has two functions:

1. Identification of the transparent IOCTLs that can be processed on a tty stream:

   The **tioc** module maintains a table containing a list of ioctls commands with the number of bytes to copy, and a type (see the **tioc_reply** structure in “Open Routine”). The **tioc** module recognizes a predefined list of ioctls. In addition, it asks the lower modules or drivers for their specific IOCTLs and adds them to its list. This protocol is described in “Open Routine.”

2. Management of the data transfers from or to the stream head for the transparent ioctls:

   Since a module or driver has no user context, it has to request the stream head to perform data transfers between user and kernel environments. The **tioc** module is responsible for the transfer of data from or to the user space. This avoids messages transfers through all modules. If the ioctl processing requires data to be transferred in from user space, **tioc** issues an **M_COPYIN** message. In the same way, **tioc** issues an **M_COPYOUT** message to transfer out any data back to user space. The **tioc** module may also intermix **M_COPYIN** and **M_COPYOUT** messages in any order, if both input and output transfers are required for an ioctl. The protocol used to perform the **M_COPYIN** and **M_COPYOUT** messages is described in “Copy in Data for an IOCTL” and “Copy out Data for an IOCTL.”

Open Routine

The open routine has to initialize the IOCTL commands table of the **tioc** module. For that, it sends downstream an **M_CTL** message containing the **TIOC_REQUEST** command.

The **M_CTL** message has to come downstream to the stream end. Then the drivers or modules send an **M_CTL** message upstream containing the **TIOC_REPLY** command. Each module on read side adds an **M_DATA** message to the **M_CTL** message if required. The **M_DATA** message contains a list of **tioc_reply** structures which define the specific IOCTLs to be handled by **tioc**.

The **tioc_reply** structure, defined in the **str_tty.h** file, is the following:

```
struct tioc_reply {
        int tioc_cmd;          /* command */
        int tioc_size;         /* number of bytes to copy */
        int tioc_type;         /* type of ioctl */
};

/*
 * STREAM tioc module tioc_type
 */
#define TTYPE_NOCOPY    0    /* don't need any copies */
#define TTYPE_COPYIN    1    /* need a M_COPYIN */
#define TTYPE_COPYOUT   2    /* need a M_COPYOUT */
#define TTYPE_COPYINOUT 3    /* need both M_COPYIN and M_COPYOUT */

#define TTYPE_IMMEDIATE 4    /* use immediate value */
```

The following examples show how IOCTL types are used.

The IOCTLs that do not require data transfer to or from STREAMS (for example, **TIOCSDTR**) are declared as TTYPE_NOCOPY.

The IOCTLs that use immediate parameter values (for example, **TCSBRK**) are declared as TTYPE_IMMEDIATE and are called as follows:

```
ioctl(fd, TCSBRK, 0);
```

The IOCTLs that request information from the STREAMS and require COPYOUT operations (for example, **TIOCGETA**) are declared as TTYPE_COPYOUT and are called as follows:

```
ioctl(fd, TIOCGETA, termios_ptr);
```

In the preceding example, `termios_ptr` is the address of a **termios** structure to be obtained from the tty.

The IOCTLs that send information to the STREAMS and require M_COPYIN operations (for example, **TIOCSETA**) are declared as TTYPE_COPYIN and are called as follows:

```
ioctl(fd, TIOCSETA, termios_ptr);
```

In the preceding example, `termios_ptr` is the address of a **termios** structure to be set in the tty.

Copy in Data for an IOCTL

When an **M_IOCTL** message arrives from the stream head, the write-side put routine recognizes an IOCTL which requires data to be copied in, and it also knows the size of required data.

So, the write-side put routine immediately sends an **M_COPYIN** message with this size to the stream head. The stream head processes a **copyin** function and sends an **M_IOCDATA** message downstream containing the data and the result of the **copyin** process.

If the **copyin** was successful, the write-side put routine sends an **M_IOCTL** message, linked to an **M_DATA** message containing the data, downstream. The **M_IOCTL** message is sent downstream until a module can process it and send an **M_IOCACK** up to the stream head.

If the **copyin** was not successful, the write-side put routine sends an **M_IOCNAK** message to the stream head.

Copy out Data for an IOCTL

When an **M_IOCTL** message arrives from the stream head, the write-side put routine recognizes an IOCTL which requires data to be copied out.

The **tioc** module sends the **M_IOCTL** message downstream and the IOCTL is processed by the appropriate module. If the ioctl processing is successful, this module sends an **M_IOCACK** upstream containing data to copy out.

The read-side put routine sends an **M_COPYOUT** message, linked to an **M_DATA** message, to the stream head. The stream head processes a **copyout** function and sends downstream an **M_IOCDATA** message containing the result of the **copyout** process.

The write-side put routine replies to this message by an upstream message, depending on the success of the **copyout**. If it was successful, it sends an **M_IOCACK** message. If not, it sends an **M_IOCNAK** message.

LDTERM Module

The **ldterm** module is a key part of the STREAMS-based tty subsystem. It supplies the line discipline for terminal devices. This line discipline is POSIX compliant and also supports the System V and BSD interfaces. The **ldterm** module provides the terminal interface functions specified in the **termios.h** header file. The **ldterm** module also handles EUC and multibyte characters.

The **ldterm** module processes various types of STREAMS messages. The messages processed by this module are listed in Messages Summary. Any other message received by **ldterm** is passed downstream or upstream unchanged.

Open Routine

When first called, the open routine allocates space for the **ldterm** internal structure, and also sends an **M_SETOPTS** message upstream. This message includes a **stroptions** structure part defined in the **stream.h** file, which contains options that inform the stream head how to use this stream.

The open routine allocates space for the **termios** structure, which contains the flags used to control the terminal. The flags are defined in the **termios.h** file.

- **c_iflag** defines input modes.
- **c_oflag** defines output modes.
- **c_cflag** defines hardware control modes.
- **c_lflag** defines terminal functions handled by **ldterm**.
- **c_cc** defines the control characters.

The open routine initializes the flags with default values.

When **ldterm** is pushed during stream initialization, it sends some **M_CTL** messages downstream that query the driver for the default flags (**TIOCGETA** comand) and the flags the driver may process (**MC_CANONQUERY** command). The driver may modify the flags processed by **ldterm** with its response to **MC_CANONQUERY**.

Close Routine

The close routine sends an **M_SETOPTS** message upstream to undo stream head changes done on the first open.

The **ldterm** module also sends **M_START** messages downstream to undo the effect of any previous **M_STOP** messages.

Finally, the close routine frees all the outstanding buffers allocated by the **ldterm** module.

Read-Side Put Routine

The **ldterm** read-side put routine processes the following STREAMS messages coming from downstream modules or driver:

M_FLUSH

This is a request to flush the read-side or the write-side queue of all its data messages and all data being accumulated. The read-side put routine processes the request and forwards the message upstream.

The queue to be flushed (read-side or write-side) is determined by the **M_FLUSH** parameter (**FLUSHR** or **FLUSHW**).

M_BREAK

The **M_BREAK** message provides the following status information:
`break_interrupt`, `parity_error`, `framing_error` or `overrun`.
The read-side put routine reads the status, processes the event and discards the message.

M_DATA

The read-side put routine processes the **M_DATA** message and performs various actions according to the characters encountered in the data and the setting of the **termios** flags:

- The read-side put routine generates echo characters which are sent downstream in **M_DATA** messages.
- **ldterm** can control the output flow of data: if the **IXON** flag is set and if **ldterm** deals with the flow control, the read-side put routine processes START (VSTART) and STOP (VSTOP) characters and sends **M_START** and **M_STOP** messages downstream.
- **ldterm** can control the input flow of data: if the **IXOFF** flag is set and input is to be stopped or started, the read-side put routine generates **M_STOPI** and **M_STARTI** messages downstream.
- If the **ISIG** flag is active, the read-side put routine manages signals characters. It sends **M_PCSIG** messages upstream when signal characters are encountered and then discards these characters.
- At the logical termination of input, the read-side put routine sends the currently buffered characters upstream to the stream head. The logical termination of input depends on the state of the **ICANON** flag:
  - If **ICANON** is set, **ldterm** is in canonical input mode. In this case, the input logically terminates at the end of a line of input. The canonical line termination characters are NEWLINE, EOF, EOL, and EOL2.
  - If **ICANON** is not set, **ldterm** is in noncanonical or raw input mode. In this case, the input terminates when at least VMIN bytes are present in the input message buffer or when the timer specified by VTIME expires.

M_IOCACK

The **M_IOCACK** message signals a positive acknowledgment of a previous **M_IOCTL** message.

- If the **M_IOCACK** message acknowledges the **TIOCGETA**, **TIOCSETA**, **TIOCSETAW**, or **TIOCSETAF** commands, the **termios** structure is updated as specified in the commands.
- If the **M_IOCACK** message acknowledges switching the current canonical mode (**–ICANON** to **ICANON**, or **ICANON** to **–ICANON)**, the read-side put routine sends an **M_SETOPTS** message upstream to notify the stream head of the change.
- If the message acknowledges a **TIOCOUTQ** command, the required number of bytes are added to the reply value in the **M_IOCACK** message.
- In all other cases the message is sent upstream.

M_CTL

The **M_CTL** messages received on the read-side and processed by **ldterm** are sent by the driver for different reasons:

1. The **M_CTL** message can be sent to communicate changes in the driver's state:

   If the **CLOCAL** flag of **ldterm** is not set, and the carrier state has just made a transition from `on` to `off`, the read-side put routine sends an **M_HANGUP** message upstream to inform the stream head that the terminal connection was broken.

2. The **M_CTL** message can be sent to answer to a previous request or command from **ldterm**. The following commands contained in an **M_CTL** message are processed:

TIOCGETA The driver sends this command either as a response to an inquiry for current settings or to reflect an asynchronous change in the flags of its **termios** structure.
The read-side put routine copies the **termios** structure from the attached **M_DATA** message block into its internal **termios** structure. Then, it frees the **M_CTL** message.

MC_NO_CANON
The input canonical processing normally performed on **M_DATA** messages is disabled and those messages are passed upstream unmodified; this is used by modules or drivers that perform their own input processing (For example a pseudo terminal in TIOCREMOTE mode connected to a program that performs the input processing).

MC_DO_CANON
All input processing performed on **M_DATA** messages is enabled.

MC_PART_CANON
The driver sends this message to notify **ldterm** that it handles some part of the input processing itself (for example, flow control). An **M_DATA** message containing a **termios** structure is expected to be attached to the original **M_CTL** message. The **ldterm** module will examine the `c_iflag`, `c_oflag`, and `c_lflag` fields of the **termios** structure and will process only those flags which have not been turned ON.

TIOCGETMODEM
The driver sends this message to communicate the state of its flag `modem carrier on`. The associated **M_DATA** message contains a value of 1 (one) to indicate the carrier is `on`, or a value of 0 (zero) to indicate the carrier is `off`. This information is used to update the **ldterm** module state. If the **CLOCAL** flag of **ldterm** is not set, and the carrier state has just made a transition from `on` to `off`, the read-side put routine sends an **M_HANGUP** message upstream to inform the stream head that the terminal connection was broken.

   When the command is processed the **M_CTL** message is freed.

3. The **M_CTL** message can be sent to answer to the **TIOC_REQUEST** coming from the **tioc** module with a **TIOC_REPLY**. (See "TIOC Module" on page 11-4.) This **M_CTL** message is forwarded upstream.

All other messages are merely forwarded upstream.

Write-Side Put Routine: Immediate Processing

The **ldterm** write-side put routine immediately processes the following STREAMS messages:

M_FLUSH

The write-side put routine flushes the read-side or the write-side queue and discards any buffered output data. Then, it forwards the message downstream.

The queue to be flushed (read-side or write-side) is determined by the **M_FLUSH** parameter (**FLUSHR** or **FLUSHW**).

M_DATA

If the write-side queue is empty, the write-side put routine processes the **M_DATA** message. Else, it queues the **M_DATA** message to the write-side queue for later processing by the write-side service routine.

M_IOCTL

The write-side put routine validates the format of the **M_IOCTL** message and checks for known IOCTL command:

- If the message format is invalid, it turns the **M_IOCTL** message into an **M_IOCNAK** message, and returns it upstream.
- If the IOCTL command is not recognized, it forwards the **M_IOCTL** message downstream for processing by other modules.
- If the IOCTL is recognized, the write-side put routine determines if the command must be processed in the proper sequence relative to **M_DATA** messages. If so, it queues the **M_IOCTL** message to the write-side queue for later processing. The commands that require processing in sequence are:
  TIOCSETAW, **TIOCSETAF**, **TCSETAW**, **TCSETAF**, **TCSBRK**, **TCSETXW**, **TCSETXF**, and **TCSBREAK**.
  Otherwise, the write-side put routine processes the command immediately.

M_READ

The **M_READ** message is processed only if the **ldterm** module is in noncanonical input mode.

The **M_READ** message is sent by the stream head to notify downstream modules when an application has issued a read request and there is not enough data queued at the stream head to satisfy the request. The message contains the number of characters requested by the application.

If VTIME is positive, the write-side put routine starts an input timer. When the timer expires, it sends all buffered input upstream.

M_START, M_STOP, M_STARTI, M_STOPI

Some IOCTLs commands (**TCXONC** with one parameter among **TCOON**, **TCOOF**, **TCION**, **TCIOF**) are issued by the application to control the flow of data. The blocked data are stored in the modules queues.

To process these IOCTLs the stream head generates and send downstream the following high priority messages:

- **M_START** to restart output of data
- **M_STOP** to stop output of data
- **M_STARTI** to restart input of data
- **M_STOPI** to stop input of data.

The **ldterm** write-side put routine updates internal state fields and forwards these messages downstream.

Write-Side Service Routine: Delayed Processing

The write-side service routine processes messages that may be delayed due to STREAMS flow control or to ioctls requiring sequential processing. The write-side service routine is called by the scheduler.

M_DATA

The write-service service routine processes the data according to the flags in the **termios** structure. It sends the processed characters downstream to the driver when the write-side queue fills up and all of the data is processed.

M_IOCTL

Some ioctl commands must wait until output drains before they are processed. **M_IOCTL** messages containing these commands are queued on the write-side queue so that the write service routine processes them in the correct sequence relative to preceeding data. The commands that require processing in sequence are:
TIOCSETAW, **TIOCSETAF**, **TCSETAW**, **TCSETAF**, **TCSBRK**, **TCSETXW**, **TCSETXF**, and **TCSBREAK**.

Multibyte Processing

The **ldterm** module handles the extended UNIX code (EUC) character encoding scheme. This encoding scheme enables the module to process multibyte characters as well as single-byte characters. It correctly handles backspacing and tabulation expansion for multibyte characters.

By default, multibyte processing by **ldterm** is turned off. When **ldterm** receives an **EUC_WSET** IOCTL call that sets character width on screen to a value greater than one, it enables multibyte processing.

When multibyte processing is turned on, the **ldterm** module automatically calls EUC routines as necessary.

Messages Summary

Messages include read-side messages and write-side messages.

Read-Side Messages

Messages processed by **ldterm**:

M_BREAK, M_CTL, M_DATA, M_FLUSH, M_HANGUP, M_IOCACK.

Messages sent by **ldterm** upstream:

M_CTL, M_DATA, M_ERROR, M_FLUSH, M_HANGUP, M_IOCACK, M_IOCNAK, M_PCSIG, M_SETOPTS.

Write-Side Messages

Messages processed by **ldterm**:

M_CTL, M_DATA, M_FLUSH, M_IOCTL, M_READ, M_START, M_STARTI, M_STOP, M_STOPI, M_NOTIFY.

Messages sent by **ldterm** downstream:

M_BREAK, M_CTL, M_DATA, M_DELAY, M_FLUSH, M_IOCTL, M_STOP, M_START, M_STOPI, M_STARTI.

SPTR Module

The serial printer module (**sptr**) supplies a specific line discipline for serial printers. It is mainly used by the spool subsystem.

The user interface (spool/application) is specified in the **lp** special file.

The **sptr** module processes various types of STREAMS messages. The messages processed by this module are listed in Messages Summary. Any other message received by **sptr** is passed downstream or upstream unchanged.

Open Routine

When first called the open routine allocates space for the **sptr** structure and sends an **M_SETOPTS** message upstream to the stream head.

When **sptr** is pushed on the stream, it sends an **M_CTL** message to the driver (containing the **TIOCMGET** command) to obtain the status and the CTS modem control signal.

Read-Side Put Routine

The read-side put routine processes the following STREAMS messages:

M_FLUSH

The read-side put routine flushes the read-side queue, then forwards the message upstream. The flushing is dependent on the input parameter (FLUSHR or FLUSHW).

M_DATA

The read-side put routine stores the **M_DATA** message for next **M_READ** message.

M_PCPROTO

The driver sends this message containing the **LPWRITE_ACK** command to indicate to **sptr** that all the data it sent in the previous **M_PROTO** message was transmitted to the line.

M_IOCACK

The **M_IOCACK** message signals the positive acknowledgment of a previous **M_IOCTL** message.

M_CTL

The **M_CTL** message is sent by the driver to communicate changes in the driver's state, or to reply to a previous **M_CTL** message.

The following commands contained in an **M_CTL** message are processed:

TIOCMGET The **sptr** module registers the CTS state in its private data.

TIOC_REPLY The **sptr** module adds information concerning the specific IOCTL commands in the message and sends it upstream.

cts_on or **cts_off**
This message is sent by the driver when the CTS signal changes. **sptr** uses it to get information about the printer connection and in turn takes appropriate actions.

Write-Side Put Routine

The write-side put routine immediately processes the following STREAMS messages: (Messages not listed here are simply forwarded downstream.)

M_FLUSH

The write-side put routine flushes the write-side queue and forwards the message downstream. The flushing is dependent on the input parameter (FLUSHR or FLUSHW).

M_DATA

If the write-side queue is not empty, the write-side put routine queues the message to this queue for later processing. The message will be processed by the write-side service routine when called by the scheduler.

If the write-side queue is empty, the write-side put routine processes the message immediately: it formats the data if needed, and sends an **M_PROTO** message with the data downstream to the driver.

M_IOCTL

The write-side put routine processes some IOCTL commands.

For example, the **sptr** module will reply to **LPWRITE_REQ** when it receives the write completion message (**M_PCPROTO** message) from the driver or if an error condition arrives (timeout, disconnection of the printer).

The IOCTL commands which are not processed by SPTR are sent downstream unchanged.

M_READ

This message is sent by the stream head as a data request. The write-side put routine returns the required number of data previously stored on the read side.

Messages Summary

Messages include read-side messages and write-side messages.

Read-Side Messages

Messages processed by **sptr**:

M_CTL, M_DATA, M_FLUSH, M_IOCACK, M_PCPROTO.

Message sent by **sptr** upstream:

M_FLUSH, M_IOCACK, M_IOCNAK, M_SETOPTS.

Write-Side Messages

Message processed by **sptr**:

M_DATA, M_FLUSH, M_IOCTL, M_READ.

Messages sent by **sptr** downstream:

M_CTL, M_DATA, M_FLUSH, M_IOCTL, M_PROTO

SLIP Module

The Serial Line Internet Protocol (**slip**) line discipline enables the TCP/IP protocol layer to use the serial lines as network interfaces.

The **slip** module is used to read internet protocol (IP) packets from a tty port, and to write IP packets to a tty port for an IP network interface.

The **slip** module reads raw data character by character from a tty port until it can assemble an IP packet and forward it for the IP network interface. In the same way, packets written to a network interface and corresponding to a tty device (**slip** interface) are written to the tty port for transmission to a remote system.

SLIP Applications

Two **slip** applications are provided: the **slattach** command and the **sliplogin** command. Both attach a tty device to a network interface.

Both the **slattach** and **sliplogin** commands configure the stream head for a tty in the following fashion:

The tty port is opened, and set to “raw” mode, with all echoing disabled. Then all active disciplines are popped from the stream for the tty, and the **slip** line discipline is pushed. For more information on **slip** configuration, see the **/etc/rc.net.serial** shell script.

The stream stack with the **slip** module is reduced to:

- The stream head
- The **slip** module
- The driver.

SLIP Routines

The **slip** module has open and close routines, and put and service routines for the read-side and the write-side.

The read-side put routine handles **M_FLUSH**, **M_CTL** and **M_DATA** messages sent by the driver. Unrecognized messages are just passed upstream.

The write-side put routine handles **M_FLUSH**, **M_IOCTL** and **M_DATA** messages sent by the stream head. Unrecognized messages are just passed downstream.

TTY Drivers

tty drivers directly control the hardware (asynchronous devices) or pseudo-hardware (pty devices). They perform the actual input and output to the adapter. The tty subsystem consists of the following drivers:

Name	Function
cxma	128-port adapter
lft	Low function terminal emulation
lion	64-port adapter
pty	Pseudo-terminal
rs	Native serial ports, 8-port and 16-port adapters.
sf	Native serial ports, in ISA bus hardware.

This section first explains how to provide a configuration routine for a tty driver. Then the open, close, write and read processings are described, as well as the open and pacing disciplines which are managed by the driver.

STREAMS tty device drivers have interrupt entry points at the hardware device interface, and have direct entry points only for the **open**, **close**, and **sysconfig** system calls. The **open** and **close** entry points are accessed via STREAMS. The stream head translates **write**, **putmsg** and **ioctl** calls into messages and sends them downstream to be processed by the driver write put routine.

Drivers Configuration Routine

In order to support dynamic loading, unloading, configuring, and unconfiguring, each tty driver must provide a configuration routine. This routine is called each time the tty driver is referenced in a load or unload operation.

Unlike an AIX non-STREAMS-based character device driver, the configuration entry point of a tty driver has does not add an entry to the device switch table. It simply declares itself to STREAMS by calling the **str_install** utility as shown in following example. The **str_install**

utility performs the internal operations necessary to add or remove the tty driver from the STREAMS internal tables.

Example

The following example is a minimal configuration routine for a tty driver called **ttyd**. Device specific configuration and initialization logic can be added as necessary. The **ttyd_config** entry point defines and initializes the **strconf_t** structure required by the **str_install** utility. In this example the **ttyd_config** operation retrieves the argument passed through the pointer specified by the **uiop** parameter. The major number is required for tty drivers and is retrieved from the **dev** parameter.

Note: Two types of DDS are handled for a tty driver:

- A DDS for the adapter contains the information common to all the lines on the adapter.
- A DDS for the line describes the attributes passed when a line is being configured on an adapter. These attributes are, for example, **TBC** (Transmit Buffer Count), **RTRIG** (Reception Trigger), default **termios** settings.

```
/* ttyd driver example:
/* BEGINNING. */

#include <sys/device.h>         /* for the CFG_* constants */
#include <strconf.h>            /* for the STR_* constants */

static struct module_info ttydm_info = {
        DRIVER_ID,"ttyd",0,INFPSZ,512,256
};

static struct qinit ttyd_rinit = {
        NULL,ttydrsrv,ttydopen,ttydclose,NULL,&ttydm_info,NULL
};

static struct qinit ttyd_winit = {
        ttydwput,ttydwsrv,NULL,NULL,NULL,&ttydm_info,NULL
};

static struct streamtab ttydinfo = {
        &ttyd_rinit,&ttyd_winit,NULL,NULL
};

ttyd_config(dev, cmd, uiop)
        dev_t   dev;
        int     cmd;
        struct  uio *uiop;

{
        struct  ttyd_dds        tmp_dds;
        static  strconf_t       ttyd_conf = {
                "ttyd", &ttydinfo, STR_NEW_OPEN, -1
         };

        if (uimove(&tmp_dds,sizeof(struct ttyd_dds),UIO_WRITE, uiop))
                return EFAULT;

        ttyd_conf.sc_major = major(dev);
        ttyd_conf.sc_sqlevel = SQLVL_QUEUEPAIR; /* for example */

        switch (cmd) {
        case CFG_INIT:return str_install(STR_LOAD_DEV, &ttyd_conf);
        case CFG_TERM: return str_install(STR_UNLOAD_DEV,
&ttyd_conf);
        default: return EINVAL;
         }
}

/*ttyd driver example:
/* END.*/
```

In this example, all the **ttyd** driver entry points are declared in the **qinit** structures that will be handled by STREAMS. These entry points are:

On the read-side:

- an **open** routine
- a **close** routine
- a **service** routine.

On the write-side:

- a **put** routine
- a **service** routine.

All these entry points are standard STREAMS interface entry points.
The only direct entry point to the driver is **ttyd_config**.

Open Disciplines

Open disciplines specify the protocol to establish a connection

Open disciplines are defined at configuration time. They can be **dtropen** (default value) or **wtopen**.

To facilitate the adding of any new discipline without modifying the driver, a module offers a common interface for open disciplines. Thus the driver doesn't have to know any information about the open discipline it invokes at open time. In the same way, the open discipline has no asumption to make concerning the driver's private data structures. An open discipline runs correctly with any driver that respects the common interface.

Driver / Common Open Discipline Module Interface

The driver has a private data structure (one per port) which contains a pointer to an open discipline. This pointer is updated at open of the current open discipline.

The driver has to:

- provide a **ddservice** routine that allows the open disciplines to consult the status of the line and to set or drop control lines.
- keep an **openRetrieve** pointer to the open discipline internal structure as long as the input routine of the open discipline is present in the input data stream. For **wtopen** this field will remain until the driver is closed. For **dtropen** it is cleared on returning from the call to the open entry of the discipline.
- notify the open discipline, if one is active, of any line status change using the **openDisc_input** entry.
- call **openDisc_open** at each open.

Interface routines

The common open discipline module interface provides the driver with five entry routines:

openDisc_open	For open discipline open
openDisc_close	For open discipline close
openDisc_input	For open discipline input
openDisc_output	For open discipline output
openDisc_service	For open discipline service

The open routine of the driver calls **openDisc_open** with:

- A pointer to a memory location. The driver will use this pointer every time it calls the open discipline routines.
- A pointer to its private data structure (one per port).

- Its **ddservice** entry point.
- An identifier of the required open discipline.
- The open status: local or remote.
- The open mode: **DNDELAY**, **NONBLOCK**,...

openDisc_open Routine

```
int     openDisc_open   (int **retrieve, caddr_t DriverPriv, int type,
                         int (* ddservice) (), int status, int mode)
{
        if      (valid open discipline type)
                return((openDisc_sw[type].open)(retrieve,
                 DriverPriv, ddservice, status, mode));
        else
                return(EINVAL);
}
```

For example, the open routine of the **dtropen** open discipline has the following form:

```
int     dtro_open       (int **retrieve, caddr_t ddpriv,
                         int (* ddservice) (), int mode, int status)
{
/* Allocate a dtro structure for that port */
        struct  dtro_t *dtro

/* Initializes the enclosed openDisc structure */

        dtro->CommonField.openDisc_next = dtro_list;
        dtro_list = dtro;       /* Chain it to existing ones */
        dtro->CommonField.DrivPriv = ddpriv;
        dtro->CommonField.ddservice = ddservice;
        dtro->CommonField.type = DTR_OPEN;

/* Pass the address of the structure to the driver, so that it can call
again */

        *retrieve = (int *) dtro;

/* Make the dtropen specific processing */
}
```

openDisc_close Routine

```
int     openDisc_close(caddr_t retrieve)
{
        return((openDisc_sw[retrieve->type].close)(retrieve));
}
```

The close routine of any open discipline must free its private structure (including **openDisc** structure) and remove it from the chain.

openDisc_input Routine

```
int     openDisc_input(caddr_t retrieve,char c,enum status s)
{
        return((openDisc_sw[retrieve->type].input)(retrieve, c, s));
}
```

openDisc_output Routine

```
int     openDisc_output(caddr_t retrieve)
{
        return((openDisc_sw[retrieve->type].output)(retrieve));
}
```

openDisc_service Routine

```
int     openDisc_service(caddr_t retrieve,enum service_commands cmd,
        void *arg)
{
        return((openDisc_sw[retrieve->type].service)(retrieve));
}
```

Pacing Disciplines

Pacing disciplines (or flow disciplines) control the flow of input and output data. The flow control can be software or hardware, depending on driver and adapter capabilities. The default mode is **xon** for a terminal.

Software Flow Control

In this case, special characters (START and STOP) indicate when the flow of data has to be stopped or resumed. **IXON**, **IXOFF** and **IXANY** flags of **termios** structure enable start and stop output or input control.

Hardware Flow Control

In this case, the flow of data is suspended or resumed by toggling an EIA modem control signal. The hardware flow control modes are: **dtr**, **rts**.

Open and Close Routines

The open routine is called whenever a device is opened. The open routine uses the open discipline specified at configuration time, and pins the private data structure.

The close routine unpins the private data structure.

During open and close, the driver is in user context such that it is able to issue sleeps and allocate dynamic memory.

The open and close routines are normally serialized by the STREAMS. Only one close can be active at a time per major/minor device pair. In some cases, the open routine can also sleep waiting for the specified conditions, such as modem status, to be met.

Write-Side Put Routine

The write-side put routine processes the following received messages:

M_BREAK, M_CTL, M_DATA, M_DELAY, M_FLUSH, M_IOCTL, M_PROTO, M_START, M_STARTI, M_STOP, M_STOPI.

Because the tty driver is the lowest module on the STREAMS stack, all other messages have to be freed by the driver.

M_DATA

Data are to be sent to ouput.

M_DELAY

Data output is suspended for a delay passed as a parameter.

M_IOCTL

The **M_IOCTL** message contains the IOCTL command to be processed by the driver. When the processing ends, the driver sends an **M_IOCACK** or **M_IOCNAK** message upstream, depending on the result of the IOCTL processing. For those IOCTLs which are not to be processed by the driver, the driver sends an **M_IOCNAK** message upstream. However some IOCTL commands are ignored by the driver and cause only an acknowledgement of the command.

M_FLUSH

If the request is done for the write side, the driver directly flushes its write-side queue. If the request is for the read side, the driver immediately sends the same message upstream and finally flushes its read-side queue.

M_CTL

The **M_CTL** message is generally sent by **ldterm**, **sptr** or **tioc** modules on their **open** processing to obtain information from the driver. It must be processed immediately. The message contains one of the following commands:

MC_CANONQUERY, **TIOCGETA**, **TIOCGETMODEM**, **TIOC_REQUEST**, **TIOCMGET**, **TXTTYNAME**.

If these commands need to be processed by the driver, the driver sends upstream the same message containing the requested information in the data part of the message.

If these commands do not need to be processed by the driver, then the driver just frees the message.

In reply to the **TIOC_REQUEST** command the driver sends the **M_CTL** message upstream replacing **TIOC_REQUEST** by **TIOC_REPLY**, and adding the list of its specific ioctl commands, if any.

In reply to the **MC_CANONQUERY** command, the driver answers **MC_PART_CANON** if needed.

TXTTYNAME is used to initialize the line discipline module name.

TIOCGETA is used to initialize the line discipline with the default **termios** settings.

M_STOP, M_START

These requests stop and restart output.

M_STOPI, M_STARTI

These requests stop and restart input.

M_BREAK

This message is sent by the line discipline to the driver to request the transmission of a BREAK on the device if it supports the break condition.

If the integer pointed by the `b_rptr` part of this message is 1, then the driver sets the break condition; otherwise the driver clears the break condition.

M_PROTO

This message is sent by the **sptr** module and contains the **LPWRITE_REQ** command. The **M_DATA** message associated with the **M_PROTO** message contains data to transmit to the line. The driver must acknowledge this message with a **M_PCPROTO** message with parameter **LPWRITE_ACK** when it has transmitted all the data to the line.

Read-Side Processing

The driver has no read-side put routine because it is the last module on the stream. However the driver has a read-side service routine which is scheduled by STREAMS.

The following messages are sent upstream by the driver:

M_BREAK, M_CTL, M_DATA, M_PCPROTO, M_PCSIG.

M_DATA

When the driver is ready to send data or other information to the user process, it does not wake up the process. It stores the received characters in **M_DATA** messages which are queued. These messages are sent later by the read-side service routine to the stream corresponding to the driver line.

M_BREAK

The driver sends this message upstream to provide the line discipline with the following status information:
`break_interrupt`, `parity_error`, `framing_error` or `overrun`.

M_CTL

The driver sends this message upstream either to communicate changes in the modem status (a transition of the carrier state from `on` to `off` for example), or to answer to a previous request (**TIOCGETA** from module **ldterm**, for example).

M_PCSIG

If the SAK recognition is set, the driver sends this message to signal the reception of the SAK sequence.

M_PCPROTO

The driver sends this message containing the **LPWRITE_ACK** command to indicate to **sptr** that all the data it sent in the previous **M_PROTO** message was transmitted to the line.

Interface with the TIOC Module

On opening, the **tioc** module sends an **M_CTL** message downstream containing the **TIOC_REQUEST** command. Then the **tioc** module waits for an **M_CTL** message with a **TIOC_REPLY** command containing the downstream modules or driver specific ioctls. The **tioc** module will update its ioctls table according to these specific ioctls. (See “TIOC Module” on page 11-4.)

Example

The **rs** driver has two transparent IOCTLs (**RS_SETA** and **RS_GETA**) which require data transfers from and into user space. The **tioc** module will perform these transfers on behalf of the **rs** driver. For that, **rs** defines two **tioc_reply** structures that will be sent to **tioc** module at its open time in reply to a **TIOC_REQUEST** included in an **M_CTL** message.

```
/* tioc_reply structures array. */

static struct tioc_reply
srs_tioc_reply[] = {
        { RS_SETA, sizeof(struct rs_info), TTYPE_COPYIN },
        { RS_GETA, sizeof(struct rs_info), TTYPE_COPYOUT },
};
```

In the write put routine of the **rs** driver, if the message to process is an **M_CTL** containing a **TIOC_REQUEST** command, the following code is executed:

```
/* mp is the M_CTL to process,
   iocp is mp->b_rptr: pointer to an iocblk struct describing M_CTL
   mp1 will contain the 2 tioc_reply structures.
   q is the driver's write-side queue.
*/

case TIOC_REQUEST:

int reply_size = 2 * sizeof(struct tioc_reply);
iocp->ioc_cmd = TIOC_REPLY;
if (!(mp1 = allocb(reply_size, BPRI_MED)))
        break;  /* just reply with the same message, next RS_SETA and
                   RS_GETA will arrive transparent and will fail */
        iocp->ioc_count = reply_size;
        bcopy(srs_tioc_reply, mp1->b_rptr, reply_size);
        mp1->b_wptr = mp1->b_rptr + reply_size;
        mp->b_cont = mp1;
        qreply(q,mp);   /* send the M_CTL upstream */
        break;
        }
```

The **M_CTL** message containing the **TIOC_REPLY** command is sent upstream and other modules will add **M_DATA** messages to **M_CTL** if necessary.

Interface with the LDTERM Module

The driver answers to the following **ldterm** commands which are included in **M_CTL** messages:

TIOCGETA The **ldterm** module asks for current **termios** structure settings.

TIOCGETMODEM
The **ldterm** module asks for the modem carrier state.

MC_CANONQUERY
The **ldterm** module negotiates which **termios** structure flags are handled by the driver.

Another specific interface between the tty driver and the line discipline module (**ldterm, sptr**) module is the possibility for the driver to send special information:

- A modem status change
- A break interrupt
- A parity error
- A framing error.

When the driver detects a modem status change, it sends upstream an **M_CTL** message with a status pointed by the `b_rptr` part of the message. This status can be:

```
cts_on, cts_off, dsr_on, dsr_off, ri_on, ri_off, cd_on, cd_off.
```

When the driver detects an error, it sends an **M_BREAK** message upstream with a status pointed by the `b_rptr` part of the message. This status can be one of the values:

```
break_interrupt, framing_error, parity_error, overran.
```

Interface with the SPTR Module

When the driver has sent on the line all the data associated with an **M_PROTO** message (with the **LPWRITE_ACK** command), it sends **sptr** an **M_PCPROTO** message (with the **LPWRITE_REQ** command) in reply.

The TTY Subsystem in a Multiprocessor Environment

Note: Information supplied in this section requires knowledge of the STREAMS synchronization, and of the device drivers in a multiprocessor environment. See Related Information on page 11-28.

On a multiprocessor system, a program can run on any processor and can migrate between processors. A program which consists of multiple threads can run on several processors at the same time. This creates a problem of concurrent access to global data.

However, the STREAMS-based tty subsystem takes advantage of the synchronization provided by STREAMS. Most of tty STREAMS modules and drivers are configured with the queue pair level synchronization (SQLVL_QUEUEPAIR). This ensures the serialization of operations done on read and write sides, without explicit locking by the module.

In the tty subsystem, the problem of maintaining data consistency in a multiprocessor environment is different in the driver and in the other stream modules.

TTY Modules Other Than Driver

If the module doesn't have global data shared by all the modules instances, the queue pair level synchronization (SQLVL_QUEUEPAIR) is enough to ensure the module is multiprocessor-safe. Nevertheless, the queue level synchronization (SQLVL_QUEUE) can be used for better throughput if there is no shared data between the read and write sides of the module, or if the put and service routines of the module guarantee the consistency of accesses to such data. The kernel provides a set of locking services and atomic primitives for that purpose.

If the module has global data, it must be protected with locks, such as simple locks. Use the **disable_lock** and **unlock_enable** kernel services to safely protect data on a multiprocessor system.

Drivers

There are three basic types of critical sections for drivers:

- thread-thread: critical sections shared between threads
- thread-interrupt: critical sections shared between threads and interrupts handlers
- interrupt-interrupt: critical sections shared between interrupt handlers.

STREAMS resolves the first critical section problem, except for concurrent multiple opens. The serialization of open and close, and the synchronization between the last close and first open is performed by STREAMS, and no specific lock is needed in the driver.

The other critical sections must be managed by the driver.

Depending on the STREAMS synchronization level selected, the STREAMS will also ensure the serialization of the put and service routines for the driver.

For the **rs** driver for example, only one interrupt per adapter can be handled at a time. But off-level routines which may be called for the same port on different processors, need to be serialized. To do so, the driver uses simple locks and interrupt priority masking.

Drivers which are not multiprocessor-safe can rely on the possibility of funnelling provided by the kernel and STREAMS.

Special Cases

These include callback functions and driver configuration routines.

Callback Functions

There are callback functions (for the **timeout** or **bufcall** utilities) that need to be protected against interrupts. This protection can be ensured by STREAMS. In this case the flag **STR_QSAFETY** will be specified in the **str_install** utility. It is also possible to use locks to protect the callback functions.

Note: These functions are not serialized with the **service** and **put** procedures.

Driver Configuration Routine

The minimal configuration routine of the driver is not called by STREAMS. Consequently it is not protected by STREAMS and must ensure its protection as a non-STREAMS driver does.

An example of a driver configuration routine is provided on page 11-14.

IOCTL Support and Origin

The following table indicates for each IOCTL:

- Its origin: AIX, AT&T, BSD or SVID (AIX means that the IOCTL is specific to AIX system).
- In which part of the stream tty subsystem (stream head, **ldterm, sptr**, **nls**, or driver) the IOCTL is processed. An asterisk (*) in a column indicates where the ioctl is processed.

The last column (comment) gives an additional information for some IOCTLs:

- “STREAMS” indicates that the ioctl is processed by the STREAMS framework.
- “TIOCGETA”, “TIOCSETA”, “TIOCSETAF” and “TIOCSETAW” indicate the internal names of the TCGETS, TCSETS, TCSETSF and TCSETSW ioctls respectively.
- “rs”, “sf”, “lion”, and “cxma” indicate that the IOCTL can only be used by the specified driver.
- “pty” indicates that the IOCTL can only be used by a pseudo-terminal driver.

IOCTL	origin	stream head	ldterm	sptr	nls	driver	comment
CXMA_GETA	AIX					*	cxma
CXMA_SETA	AIX					*	cxma
CXMA_SETAW	AIX					*	cxma
CXMA_SETAF	AIX					*	cxma
CXMA_KME	AIX					*	cxma
CXMA_GETFLOW	AIX					*	cxma
CXMA_SETFLOW	AIX					*	cxma
CXMA_GETAFLOW	AIX					*	cxma
CXMA_SETAFLOW	AIX					*	cxma
CXMA_RESET	AIX					*	cxma
EUC_WGET	AT&T		*				
EUC_WSET	AT&T		*				
FIOASYNC	BSD	*					STREAMS
FIONREAD	BSD	*					STREAMS
LI_GETVT	AIX					*	lion
LI_SETVT	AIX					*	lion
LI_GETXP	AIX					*	lion
LI_SETXP	AIX					*	lion
LI_SLPI	AIX					*	lion
LI_DSLP	AIX					*	lion
LI_SLPO	AIX					*	lion
LI_PRES	AIX					*	lion, cxma
LI_DRAM	AIX					*	lion, cxma
LI_GETTBC	AIX					*	lion
LI_SETTBC	AIX					*	lion
LPRGET	AIX			*			
LPRSET	AIX			*			

IOCTL	origin	stream head	ldterm	sptr	nls	driver	comment
LPRMODG	AIX			*			
LPRMODS	AIX			*			
LPRGOTV	AIX			*			
LPRSTOV	AIX			*			
LPQUERY	AIX			*			
LPRGETA	AIX			*			
LPRSETA	AIX			*			
LPWRITE_REQ	AIX			*			
RS_GETA	AIX					*	rs, sf
RS_SETA	AIX					*	rs, sf
TCFLSH	SVID	*					
TCGETA	SVID		*	*			
TCGETS	SVID		*			*	TIOCGETA
TCGETX	SVID		*			*	
TCGMAP	AIX				*		
TCKEP	AIX	*					STREAMS
TCLOOP	AIX					*	
TCQSAK	AIX					*	
TCQTRUST	AIX	*					
TCSAK	AIX					*	
TCSBRK	SVID					*	
TCSBREAK	AIX					*	
TCSETA	SVID		*	*		*	
TCSETAF	SVID		*	*		*	
TCSETAW	SVID		*	*		*	
TCSETS	SVID		*			*	TIOCSETA
TCSETSF	SVID		*			*	TIOCSETAF
TCSETSW	SVID		*			*	TIOCSETAW
TCSETX	SVID		*			*	
TCSETXF	SVID		*			*	
TCSETXW	SVID		*			*	
TCSMAP	AIX				*		
TCTRUST	AIX	*					
TCXONC	SVID	*					
TIOCCBRK	BSD		*				
TIOCCDTR	BSD					*	
TIOCCONS	AIX	*					
TIOCEXCL	BSD					*	pty
TIOCFLUSH	BSD		*				
TIOCGETC	BSD		*				
TIOCGETD	BSD		*				

IOCTL	origin	stream head	ldterm	sptr	nls	driver	comment
TIOCGETP	BSD		*				
TIOCGLTC	BSD		*				
TIOCGPGRP	SVID	*					
TIOCGSID	SVID	*					
TIOCGWINSZ	BSD		*				
TIOCHPCL	BSD		*				
TIOCNXCL	BSD					*	pty
TIOCLBIC	BSD		*				
TIOCLBIS	BSD		*				
TIOCLGET	BSD		*				
TIOCLSET	BSD		*				
TIOCMBIC	SVID					*	
TIOCMBIS	SVID					*	
TIOCMGET	SVID		*			*	
TIOCMSET	SVID		*			*	
TIOCOUTQ	BSD		*			*	
TIOCPKT	BSD					*	pty
TIOCREMOTE	AT&T					*	pty
TIOCSBRK	BSD		*				
TIOCSDTR	BSD					*	
TIOCSETC	BSD		*				
TIOCSETD	BSD		*				
TIOCSETN	BSD		*				
TIOCSETP	BSD		*				
TIOCSLTC	BSD		*				
TIOCSPGRP	SVID	*					
TIOCSTART	BSD	*					
TIOCSTI	BSD		*				
TIOCSTOP	BSD	*					
TIOCSWINSZ	BSD		*			*	pty
TIOCUCNTL	BSD					*	pty
TXGPGRP	AIX	*					
TXISATTY	AIX	*					
TXSETIHOG	AIX		*				
TXSETOHOG	AIX		*				
TXSPGRP	AIX	*					
TXTTYNAME	AIX					*	

TTY Data Structures

The following is an extract from the **usr/include/sys/str_tty.h** file. This file defines all the constants, functions, structures and types of messages that are used by the tty modules and drivers for the exchange of messages. Additional comments explain how some of the structures are used.

Information from usr/include/sys/str_tty.h

```
#ifndef         _H_STR_TTY
#define         _H_STR_TTY

#include        <sys/termio.h>
#include        <sys/stream.h>
#ifdef          _KERNEL
#include        <sys/errno.h>
#include        <sys/ioctl.h>   /* for TTNAMEMAX definition */
#include        <termios.h>
#include        <sys/trchkid.h>
#include        <sys/atomic_op.h>

/* Macro definition for atomic operation on sysinfo fields. */
#define         sysinfo_add(x,y)        fetch_and_add(&(x), (y))
```

TIOC Module

```
/* Commands of the M_CTL message at open */

#define TIOC_REQUEST    _IO('J', 0x91)  /* request for ioctls */
#define TIOC_REPLY      _IO('J', 0x92)  /* reply for ioctls */

/* Data structures for TIOC_REPLY */

struct  tioc_reply {
        int     tioc_cmd;       /* ioctl command */
        int     tioc_size;      /* number of bytes to copy */
        int     tioc_type;      /* type of ioctl */
};

/* Values for tioc_type */
```

tioc_type indicates if the IOCTL requires data to be copied into or from user space.

```
#define TTYPE_NOCOPY    0       /* don't need any copy */
#define TTYPE_COPYIN    1       /* need an M_COPYIN */
#define TTYPE_COPYOUT   2       /* need an M_COPYOUT */
#define TTYPE_COPYINOUT 3       /* need both M_COPYIN and M_COPYOUT */
#define TTYPE_IMMEDIATE 4       /* use immediate value */
```

TTY Commands Associated with M_CTL Messages

```
#define TIOCGETMODEM    _IO('J', 0xa0)  /* get the modem state from driver */
#define MC_CANONQUERY   _IO('J', 0xa1)  /* query the termios state */
#define MC_NO_CANON     _IO('J', 0xa2)  /* set pty's remote mode */
#define MC_DO_CANON     _IO('J', 0xa3)  /* reset pty's remote mode */
#define MC_PART_CANON   _IO('J', 0xa4)  /* oflag,iflag and lflag of termios*/
                                        /* are handled by driver or ldterm */
```

M_PCPROTO Commands

```
#define LPWR            ('l'<<8)
#define LPWRITE_ACK     (LPWR|31)       /* command in M_PCPROTO message */

typedef int     OSR_STATUS;
```

Status Information

`enum status` gives general status definitions for drivers and line discipline in the case of parity and framing error or break_interrupt from the adapter, or in the case of modem status changes. The status information is sent by the driver in an **M_BREAK** message.

```
enum status {
        good_char, overrun, parity_error, framing_error, break_interrupt,
        cts_on, cts_off, dsr_on, dsr_off, ri_on, ri_off, cd_on, cd_off };
```

The possible values for the status enumeration are:

good_char A valid character was received.

overrun Characters were not removed from the hardware in a timely fashion and some data was lost by the hardware.

parity_error Character was received with improper parity. The character is passed as received by the hardware.

framing_error Character was received with a framing error. This usually indicates the number of bits per character is set incorrectly or the baud rate is not set correctly. The character is passed as received by the hardware.

break_interrupt
The hardware detected a break condition. The break condition is different for various adapters and physical link layers. For asynchronous communications, the break condition is usually defined as a spacing condition on the line for more than one total character time.

cts_on The clear to send signal, **cts**, made a low to high transition.

cts_off The clear to send signal made a high to low transition.

dsr_on The data set ready signal, **dsr**, made a low to high transition.

dsr_off The data set ready signal made a high to low transition.

ri_on The ring indicate signal, **ri**, made a low to high transition.

ri_off The ring indicate signal has made a high to low transition.

cd_on The data carrier detect signal, **cd**, made a low to high transition.

cd_off The data carrier detect signal made a high to low transition.

TTY Trace Support

The main tty hook IDs are defined in the **sys/trchkid.h** common header file. The tty subhooks are defined below. The tty hooks and subhooks are used in the **Return** and **Enter** macros defined below.

```
/* TTY subhooks identificators */

#define TTY_CONFIG      0x01
#define TTY_OPEN        0x02
#define TTY_CLOSE       0x03
#define TTY_WPUT        0x04
#define TTY_RPUT        0x05
#define TTY_WSRV        0x06
#define TTY_RSRV        0x07
#define TTY_REVOKE      0x08    /* for stream head */
#define TTY_IOCTL       0x09    /* for ioctls   */
#define TTY_PROC        0x0a    /* for drivers  */
#define TTY_SERVICE     0x0b    /* for drivers  */
#define TTY_SLIH        0x0c    /* for drivers  */
#define TTY_OFFL        0x0d    /* for drivers  */
#define TTY_LAST        0x0e    /* can be used for any specific entry*/
```

The parameters for the **Return** and **Enter** macros are:

w	`TTY hookid | TTY subhookid`
dev	`dev(type dev_t)`
ptr	address of the private data (`q->q_ptr`) of each module or driver
a, b, c	specific parameters for each subhook
retval	return value

The parameters for each subhook are:

TTY_CONFIG	a=command, no dev, no ptr.
TTY_OPEN	a=oflag, b=sflag
TTY_CLOSE	a=flag
TTY_WPUT	a=@msg, b=message type
TTY_RPUT	as TTY_WPUT
TTY_WSRV	a=q_count
TTY_RSRV	as TTY_WSRV
TTY_REVOKE	a=flag
TTY_IOCTL	a=ioctl command
TTY_PROC	a=cmd, b=arg
TTY_SERVICE	a=service command, b=arg
TTY_SLIH	no dev, ptr=@struct intr, a=adapter type
TTY_OFFL	ptr=@struct intr

Enter and Return Macros

The **Enter** and **Return** macros use the tty hooks and subhooks.

```
#define Enter(w, dev, ptr, a, b, c)             \
        dev_t   DEV;                            \
        int     PTR;                            \
        int     Flag = 0;                       \
        int     W;                              \
        if (TRC_ISON(0)) {                      \
                DEV = (dev);                    \
                PTR = (ptr);                    \
                Flag = 1;                       \
                W = w;                          \
                TRCHKGT(W, DEV, PTR, a, b, c);  \
        }

#define Return(retval) {                        \
        int     RET = (retval);                 \
        int     Line = __LINE__;                \
        if (Flag)                               \
                TRCHKGT(((W)|0x80), DEV, PTR, RET, Line, 0);\
        return(RET);                            \
}

#define Returnv(retval) {                       \
        int     Line = __LINE__;                \
        if (Flag)                               \
                TRCHKGT(((W)|0x80), DEV, PTR, 0, Line, 0);\
}

#define Data(xxx, a, b, c)                      \
        (Flag ? TRCHKGT(((W)|0x40), DEV, PTR, a, b, c) : 0)
```

TTY IOCTL Internal Names

The following definitions indicate the internal names of some IOCTLs.

```
/* tty ioctls commands internal names */

#define TIOCGETA        TCGETS
#define TIOCSETA        TCSETS
#define TIOCSETAW       TSETSW
#define TIOCSETAF       TCSETSF
```

STREAMS TTY Modules and Drivers DDS

```
/* Maximum device name length */

#define DEV_NAME_LN     16

/* DDS types used at configuration time */

enum dds_type {         /* Which DDS type */
        LC_SJIS_DDS,    /* sjis lower converter module*/
        LDTERM_DDS,     /* ldterm module */
        LION_ADAP_DDS,  /* 64-port driver (for adapter) */
        LION_LINE_DDS,  /* 64-port driver (for lines) */
        NLS_DDS,        /* nls module */
        PTY_DDS,        /* pty module */
        RS_ADAP_DDS,    /* Native 8 and 16-port driver (for adapters) */
        RS_LINE_DDS,    /* Native 8 and 16-port driver (for lines) */
        SPTR_DDS,       /* sptr module */
        TIOC_DDS,       /* tioc module */
        UC_SJIS_DDS     /* sjis upper converter module */
        CXMA_ADAP_DDS   /* 128-port driver (for adapter) */
        CXMA_LINE_DDS   /* 128-port driver (for lines) */
  };
```

STREAMS TTY Modules and Drivers Names

```
enum module_names {
        tioc,           /* transparent ioctl modulename */
        ldterm,         /* line discipline module name */
        pty,            /* pseudo tty driver module name */
        uc_sjis,        /* upper converter sjis module name */
        lc_sjis,        /* lower converter sjis module name */
        nls,            /* mapping discipline module name */
        sptr,           /* serial line printer discipline name */
        rs,             /* rs driver name */
        lion,           /* lion driver name */
        cxma            /* 128 port driver name */
 };

#endif  /* _H_STR_TTY */
```

Related Information

Discussion of Multiprocessing (MP) Serialization in Serialization Services, on page 5-8, and MP-Safe Coding Example, on page 5-12.

The TTY Subsystem in *AIX Version 4.1 General Programming Concepts: Writing and Debugging Programs*.

STREAMS in *AIX Version 4.1 Communications Programming Concepts*.

UNIX System V Release 4, Programmer's Guide: STREAMS. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1990.

Chapter 12. Implementing Graphical Input and 2D Graphics Device Drivers

This chapter provides technical guidance for developers who want to add functionality into the XServer on AIXwindows. This additional functionality may include graphics adapters, input devices and dynamically loaded extensions.

This chapter has several sections:

- "Porting to the AIXwindows X Server: Overview," on page 12-1, provides a description of the basics of porting to the AIXwindows X Server.
- "Porting 2D Graphics Adapters," on page 12-2, outlines the general procedure to develop a graphics adapter device driver and the loadable DDX interface used by the X Server.
- "Porting Input Devices," on page 12-47, describes how to add a new input device to the AIXwindows X Server via the X11 Input Extension.
- "Building a Dynamically Loadable Module," on page 12-60, is a brief description of how to develop a generic X extension load module.

The information provided in this chapter does not attempt to educate readers about the X Window System or X programming. Rather, it describes the tasks required to implement the above-mentioned functionality in the AIX system. For general information about the X Window System programming see the list of related information at the end of this chapter.

Porting to the AIXwindows X Server: Overview

You can add functionality to the AIXwindows X server through the creation of dynamically loadable modules. Each load module must provide the following basic functionality:

- An entry point definition
- A set of routines to initialize the appropriate data structure provided by the X server
- X Consortium and vendor-written implementation-specific routines

The load modules can interact with the X Window System in a variety of ways. The AIXwindows X server currently supports the following functionality:

- Porting the X server to run on a different 2D graphics adapter
- Writing load modules to the standard X11 Input Extension
- Creating your own dynamically loadable X Window System extension

The load modules for 2D graphics adapters and extension input devices require that entries be defined and configured in the AIX Object Data Manager (ODM) databases.This allows the X server to determine where the load modules for the specific devices reside. Information about 2D graphics adapters is stored in the ODM Graphics Adapter Interface (GAI) database, and information about extension input devices is stored in the ODM XINPUT database.

Current database entries can be examined by setting the environment variable **ODMDIR=/usr/lib/objrepos** and running the command `odmget GAI` or `odmget XINPUT`. These entries will be explained in greater detail in the respective sections of this chapter.

The dynamically loaded extensions are normally loaded via a per-user basis. If the X server is started with a `-x ext_name` flag, then the X server will look in the file **/usr/lpp/X11/bin/dynamic_ext** for the load module that corresponds to `ext_name`. There are also static extensions, which can be found in **/usr/lpp/X11/bin/static_ext**, but they are always loaded into the X server, and are thus not recommended.

The information provided here assumes an ability on the part of the developer to program generic actions without further instruction. Refer to the following sections to learn more about extension-specific tasks:

- Porting 2D Graphics Adapters (on page 12-2)
- Porting Input Devices (on page 12-47)

The subroutines used in these extensions are summarized in “List of X Server Porting Subroutines” on page 12-62.

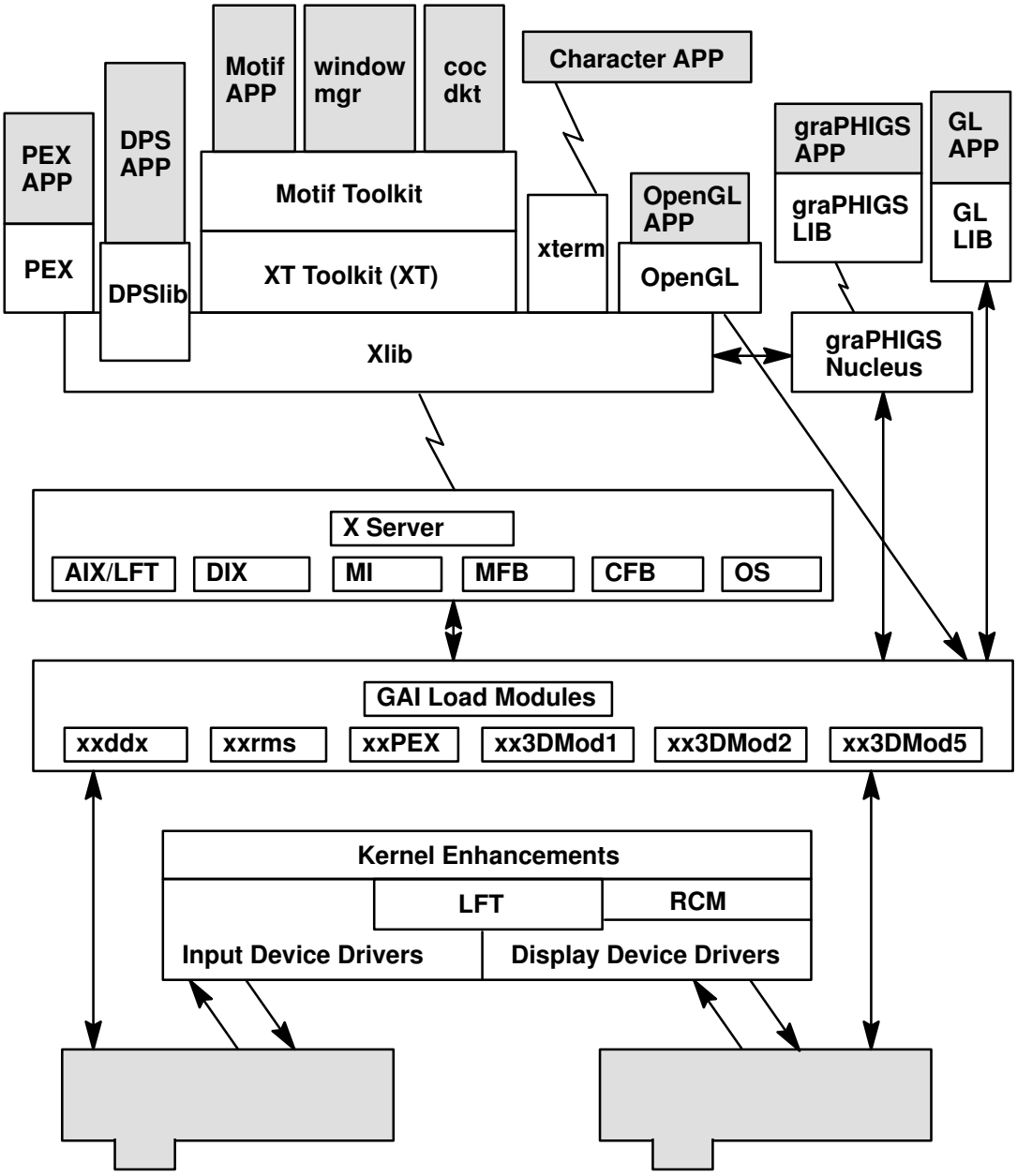
Porting 2D Graphics Adapters

This section describes the method for porting 2D graphics adapters into the X server on the AIXwindows. The information is divided as follows:

- Graphics Architecture Overview
  - Graphics Adapter Interface (GAI) Display Subsystem
  - X Server
- Low-Level Display Driver
  - Display Device Driver
  - Display Device Driver Subroutines
  - LFT Interface Routines
  - Display Driver Structure Descriptions
- Device Dependent Driver (DDX)
  - GAI 2D Adapter Load Modules
  - X Server Initialization Routines
  - Device Dependent Initailization Subroutines
  - Adapter Access and the aixgsc System Call
  - Minimum RMS for 2D Adapters
  - Configuring the 2D Adapter into the ODM Database

Graphics Adapter Interface (GAI) Display Subsystem

The Architecture of the Display Subsystem figure illustrates the display subsystem architecture.



Architecture of the Display Subsystem

The display subsystem is the set of software entities that provides all display related input and output to applications and to the AIX Version 4.1 kernel. Its capabilities range from simple 2D graphics to support for 3D models which enable lighting and shading.

A Graphics Adapter Interface (GAI) defines an interface between the user application programming interface (API) and the device-specific code.

The principal components of the architecture include:

- X Server
- graPHIGS Nucleus

- Kernel Enhancements
- GAI Load Modules

The shaded objects in the architecture figure show input and graphics hardware devices and user applications. These are not components of the display subsystem.

The components of the display subsystem receive their external inputs from X clients and from other kernel subsystems. User inputs are processed by the kernel and routed through the X server, which sends them appropriately to X clients.

Applications use programming interfaces made available through the various libraries of the display subsystem. Each programming interface, (API), defines a graphical model. These graphical models evolve in many forums, including the American National Standards Institute, the International Standards Organization, the X Consortium, the GL Architecture Review Board (ARB), and various highly successful industry *de-facto* standards.

The graphical models supported by the display subsystem are listed below:

- graPHIGS
- X Window System
- Display PostScript
- Graphics Library of Silicon Graphics, Inc. (GL)
- OpenGL Version 1.0
- PEX 5.1

Display Subsystem Definitions

The display subsystem is the set of software units that provides all display-related input and output to applications and to the AIX Version 4.1 kernel.

List of Component Types

The types of components are defined below:

Component Type	Definition
Adapter	With reference to input devices, an adapter specifies the device driver used to process user events from one of the input devices. This could be the keyboard/sound, mouse, tablet, dials, or lighted programmable function keys.
Application	Specifies a program that uses the display subsystem. The use of this term with respect to the display subsystem is analogous to the standard definition of application.
Client	Specifies an application that uses the services of X Window System. In the terminology of the display subsystem, however, this concept has been extended. A client is an application that uses a server. These clients are typically different processes and may reside on different platforms.

Display Device Driver

Specifies the set of subroutines that control read or write access to a display adapter. The display drivers provide a common programming interface to other components in the display subsystem, and account for the particular function and implementation within the specific device. Thus, each display driver contains code unique for that device. A set of subroutines is not considered a display driver unless it:

- Contains software that is applicable only to the function or control of a particular device.

OR

- Issues I/O level commands to specifically manipulate the device hardware.
- Manages the graphics context and graphics resources on behalf of servers.

Library Specifies a programming interface consisting of a functionally related set of subroutines that are typically grouped as one module and that provide the formal application programming interface into a collection of software.

Nucleus Specifies a server. The term nucleus is used principally to distinguish an instance of a server process from the X server. The principal user of nucleus is the graPHIGS API, via the graPHIGS nucleus.

Server Specifies the component of X Window System that receives requests from the Xlib library and that returns events to the Xlib library. This concept has been modified and extended in the display subsystem. A server can also mean a process executing in a platform in which rendering occurs and that is directly attached to the display adapter. This server accepts X protocol requests and returns X protocol events, and as such is the X server. However, other graphical models use the server to obtain screen resources such as window geometries or colormaps, so the X server definition is now extended to become the resource server.

Toolkit Specifies a library of calls that allows applications to manipulate data presentation using a different level of abstraction than that offered by an underlying library. The display subsystem expansion of this concept of a toolkit is the concept that it in turn calls upon another library. The toolkit often takes its name from the underlying library. Frequently, toolkits are written to be window system independent.

List of Component Names

The following list introduces the components and provides an description of the function of each. Where appropriate, the general category of component is described rather than each instance. Some of the components listed (such as DPS, GL and graPHIGS) are packaged into separate LPPs and must be ordered as such.

Component	Description
DPS	Specifies Display PostScript, a program product from Adobe Systems, Inc. This program implements the language PostScript and its extensions, as applicable, on a display system instead of on printers. DPS uses 2D graphical functions and complex character fonts, and operates as an extension of the X server.

GAI Specifies the Graphics Adapter Interface, including the interface to and the set of libraries that form the interface for graphical output to the display adapters. The GAI libraries implement 2D and 3D graphics functions as well as permit direct access to the hardware. GAI components include:

- GAI Resource Management Support
- GAI 2D Drawing Library
- GAI 3D Model 1 Drawing Library
- GAI 3D Model 2 Drawing Library
- GAI 3D Model 5 Drawing Library

GL Specifies Graphics Library. This it the library of graphical functions defined and implemented by Silicon Graphics, Incorporated, under the product name GL. This library is a set of 3D functions, based on a graphical model differing from the PHIGS graphical model. GL is the second graphical model used for 3D. Thus, it is often referred to as the 3D-M2 or the GAI 3D Model 2 Drawing Library.

graPHIGS Specifies the graPHIGS application programming interface (API), which is an implementation of PHIGS. This program includes extensions to and deviations from the PHIGS standard. The term PHIGS is a reference to the graPHIGS API, unless explicitly stated to the contrary.

The graPHIGS API is the first 3D graphical model. It is referred to as 3D-M1 or the GAI 3D Model 1 Drawing Library.

OpenGL Specifies OpenGL, a network transparent API for developing applications using 3D graphics. OpenGL is derived from the proprietary SGI GL API.

PEX Specifies the 3D Extension to X. The core X protocol provides for basic 2D graphics functionality. The PEX extension adds 3D graphics at about the same functional level as PHIGS and PHIGS PLUS, including features such as lighting, shading, server-side stored structures, and advanced primitives.

PHIGS Specifies Programmer's Hierarchical Interactive Graphics System, an accepted international standard (ISO 9592) for the definition, display, and modification of 2D or 3D graphical data, the additional manipulation of geometrically related, objects, and the definition and modification of the relationships between the data and the objects. The relationships and the data are stored in a hierarchical data store.

LFT Specifies low-function terminal. The LFT is a STREAMS-based simple character oriented tty-like terminal emulator for full screen processing. LFT is a low-cost, low-function enablement feature intended to be used only during system startup, installation and standalone diagnostics. It is not expected to be used in steady-state processing. It supports all the display adapters and keyboards available in AIX Version 4.1, but does not support the mouse, tablet, or any other input devices.

X Window System

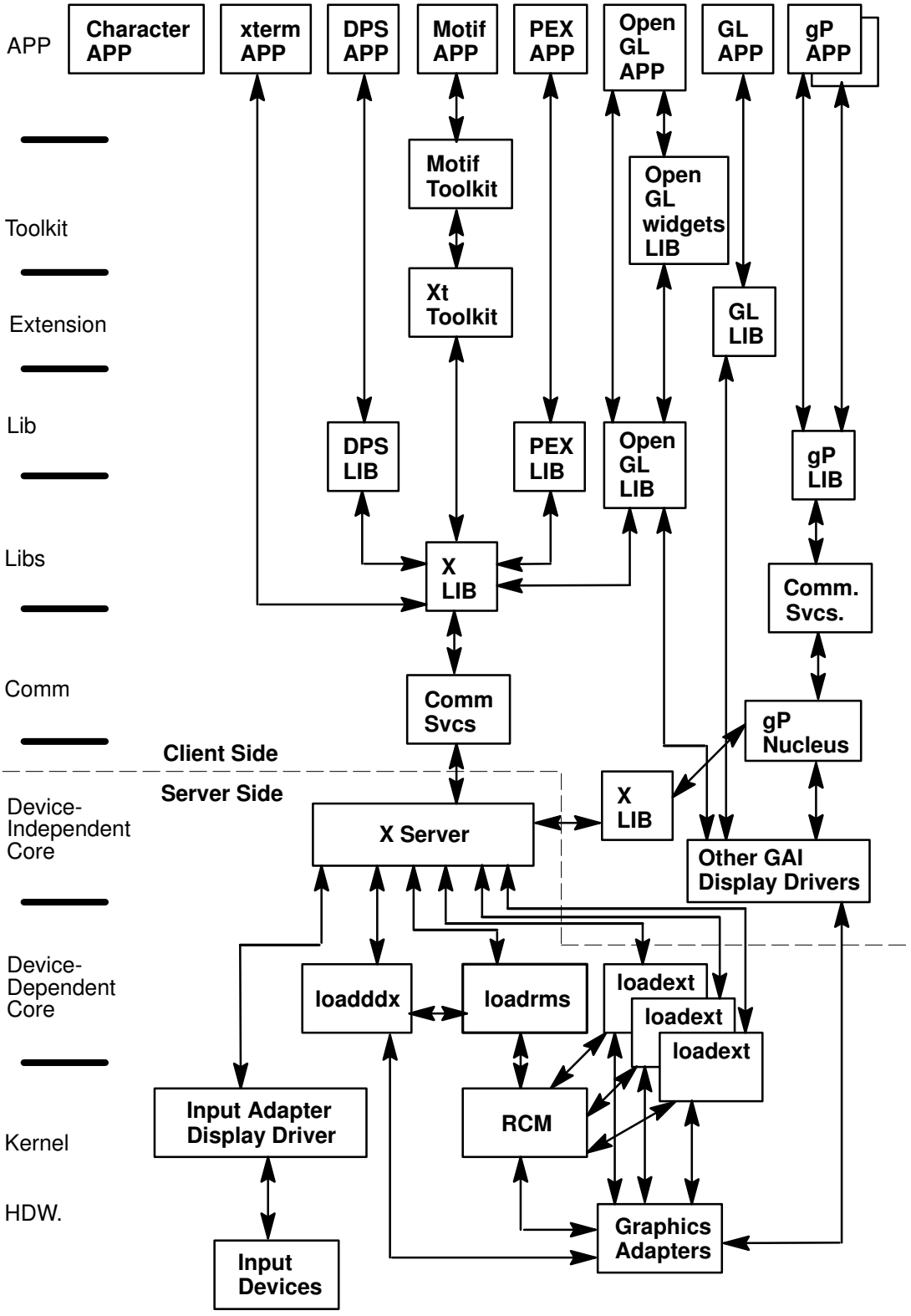
Specifies the core of the display subsystem. The X Window System supplies many functions to other components of the subsystem, including resource management, window allocation, a transparent method for communications between platforms, and 2D drawing operations. It has a well-defined mechanism for extending its services. It operates using two X components, the Xlib library and the X server that operate as follows:

Xlib Resides in a client and provides the API to users of X Window System. Xlib passes the requests to the X server and events to the application. It interacts with the X server using a protocol through an AIX or shared memory socket.

X Server

The X server resides in the X executable and is a component within the display subsystem core. The X server consists of two parts. A part that is device independent (dix) that interprets requests from the Xlib, schedules client activity, manages the return of events and input to the Xlib Library, and performs other generic actions.

The X server also consists of a device-dependent part. The device-dependent part renders the 2D graphics operations defined by X Window System for the specific display adapter. The **loadddx** GAI Load Modules implement this interface.



Functional Block Diagram of Display Subsystem

Application Programming Interface (API)

All types and combinations of applications may exist within a platform. Many of them communicate with the user and display adapter in the server by means of the X Window System protocol over a communications link. Others communicate by means of a graPHIGS protocol over (perhaps a different) communications link.

A list of the generic application types include:

- Character-based applications
- Terminal emulator applications (such as xterm)
- Toolkit applications (Motif and Xt)
- Display PostScript applications
- X Window System applications
- OpenGL applications
- GL 3.2 applications
- graPHIGS applications
- PEX applications

Applications *bind* to their appropriate library using any of the AIX system services available to them. The API component provides a set of libraries and toolkits for use by application developers.

API Names		
Common Name	**Library Name**	**Description**
Xlib	libX11.a	Low-level X11 interface. Interface to communication services.
Xt	libXt.a	Toolkit intrinsics provided by X Window System.
Xext	libXext.a	X Window System Extension libraries. Provides API for the shape, cursor, colormap, DPS, Direct Access.
Input Library	libXi.a	Provides API for the standard X11 Input Extension.
Motif toolkit	libXm.a	Provides Motif widget and API support.
Motif Resource Manager	libMrm.a	Motif Resource Manager.
OpenGL	libGL.a	Provides OpenGL support.
OpenGLwidgets	libXGLW.a	Provides support for OpenGL widgets.
OpenGL utilities	libGLu.a	Provides a set of utilities to be used with OpenGL.
GL3.2	libg1.a	Provides support for GL 3.2. Sometimes the math library is also needed with GL 3.2.
PEXlib	libPEX5.a	PEXlib API.
PEX PHIGS	libphigs.a	PHIGS API to PEX. This is sometimes called PEX_SI PHIGS.
graPHIGS lib	libgP.a	graPHIGS shell (API bindings and interface to the Nucleus)

The API and library architecture is shown in the Client Side portion of the Functional Block Diagram of Display Subsystem figure, on page 12-8.

All communications begin with AIX sockets. In local or standalone platforms, the sockets are local domain or in shared memory. Between distributed platforms, the sockets are TCP/IP sockets. The type of communication or transport medium is transparent to the application.

X Server

The X server plays an extremely important role within the display subsystem architecture. The X server is the central arbiter and provider of graphical resources. It can be considered the resource server of the display subsystem. The family of X server extensions rely upon the X server. All input flows through the X server. Other components, such as the graPHIGS nucleus or one of the 3D libraries, request basic graphics resources from the X server. The X server's role in the display subsystem architecture is shown in the Functional Block Diagram of Display Subsystem figure, on page 12-8.

The subcomponents of the server are:

- Device Independent Core
- GAI Load Modules—loadddx, loadrms
- X Extensions

The X server communicates with the applications to present data to the screen. It has an input and an output path. *Input* is defined as the direction of data flowing from an input device, through the server, and out through the communications services to the application. *Output* is defined as the direction of data flowing from an application, through the communications services, through the server and device drivers, and onto the display adapter.

The device-independent portion of the X server is made up of modified code from the X Consortium. Specifically the dix (device-independent x), mi (machine independent), mfb (monochrome frame buffer), cfb (color frame buffer) and os (operating system) directories are included. The structure and subroutines have been modified to provide for windows that are being accessed directly by one of the 3D GAI models. In other cases, modifications have been made to optimize for performance. However, in all cases, the programmer interfaces have remained the same to facilitate porting from other platforms.

The device-dependent portion of the X server is provided by the 2D GAI load modules (loadddx and loadrms). The X Window System has a regular architecture to support the definition of extensions. New and third-party extensions will be permitted and assisted.

The following extensions are supported in AIXwindows:

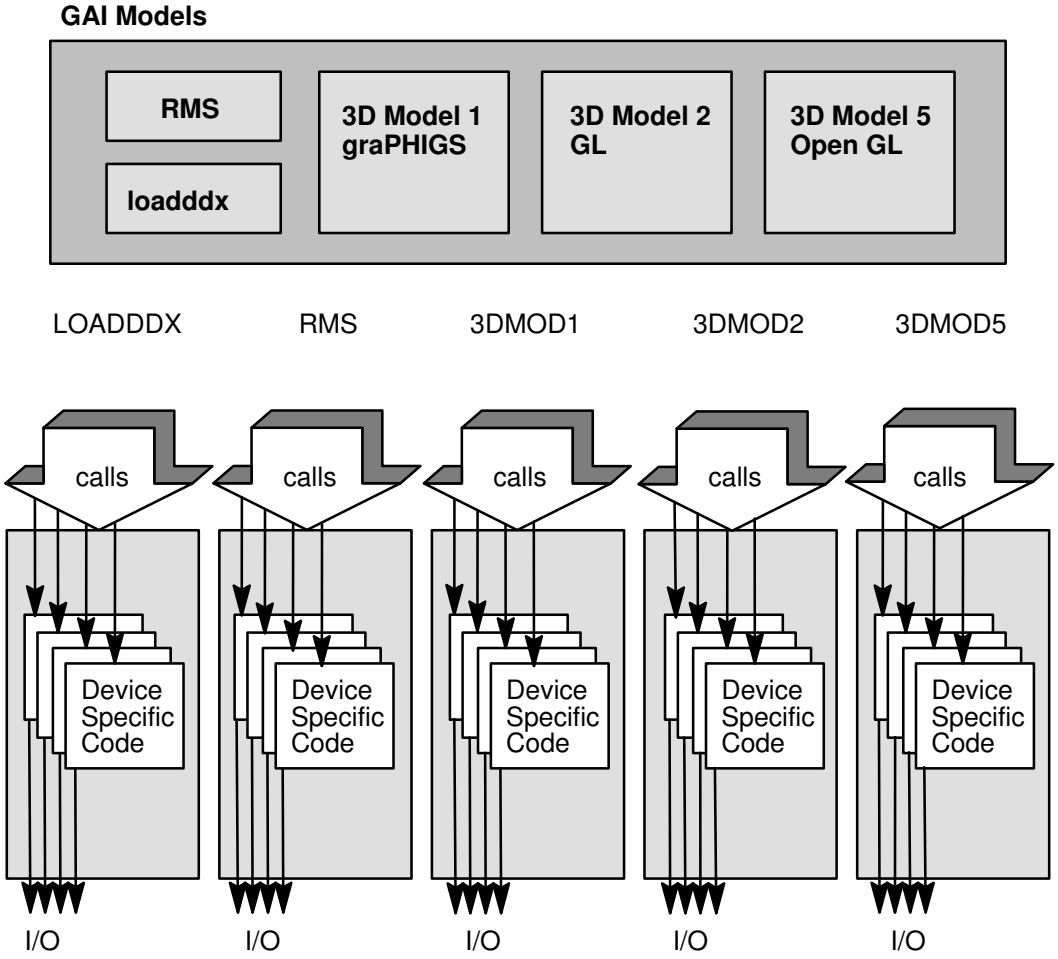
X Window System Extensions		
Common Name	**Official Name**	**Source**
DPS	Adobe-DPS-Extension DPSExtension	Adobe
PEX	X3D-PEX	MIT Standard
OpenGL	OpenGL	OpenGL Architecture Review Board
cursor	aixCursorExtension	AIXwindows
colormap blink	xColormapExtension	AIXwindows
X Input	XInputExtension	MIT Standard
Shape	SHAPE Non Rectangular Window (Shape) Extension	MIT Standard
Screen Saver	SCREEN-SAVER	MIT Draft Standard

The X Window System extensions have subcomponents in both the X server and API display system components.

GAI Load Modules

One of the fundamental components of the display subsystem is the set of graphics services provided X servers, the graPHIGS nucleus or one of the 3D libraries. These services are contained within the GAI load modules. The GAI load module is bound to the X server, graPHIGS nucleus or 3D library upon its initialization. It provides a library of graphics function to the loading entity.

Except for the RMS load modules, the GAI load modules each provide the device-specific code for one of the GAI models, as illustrated in the Block Diagram of the GAI Load Modules figure. RMS provides a set of support routines that manage hardware resources common to all models, such as windows and clipping regions, but not all models are supported on all adapters. Consequently, there is be no need for full function RMS on this class of adapter.



Block Diagram of the GAI Load Modules

The GAI load module directly controls the display adapters. When called, it passes direct I/O operations to the display adapters.

The GAI load module also provides functions that assist the X server and other servers in implementing a protocol to directly access the hardware and in managing the resources of the display adapters. It also processes inputs from the display adapters, particularly interrupts and pick events.

The GAI load modules are as follows:

loadddx The loadddx load modules implement the porting layer of the X server. Its interfaces can be found in *Strategies for Porting the X v11 Sample Server*. There is one loadddx for each graphics adapter supported. “Graphics Adapter Interface (GAI) 2D Adapter Load Modules,” on page 12-33, provides techniques for the development of a loadddx.

loadrms The loadrms load modules implement the resource management support for the GAI models. This includes device specific management of window geometries, windows, monitors, system queues, and clipping regions. This resource management is most important for the 3D models accessing hardware directly. “Minimum Resource Management Subsystem (RMS) for 2D Adapters,” on page 12-44, discusses the minimum RMS support needed to implement a 2D graphics adapter.

Note: The following load modules are necessary to provide certain types of 3D support to adapters. They are **not** required for 2D adapters, but are listed here only for informational purposes.

load3dm1 The GAI 3D Model 1 load modules provides the set of functions required to support a PHIGS-like 3D graphics model within the display subsystem. The support functions have been tailored to complement the architecture of the graPHIGS Nucleus. The functions supported fall into the broad categories of context support and rendering support. The 3dm1 load module provides the support necessary for multiple graPHIGS nucleii to access a display adapter.

load3dm2 The API within the display subsystem that supports immediate mode graphics is GL. To support this API, GAI 3D Model 2 load module has been defined. It provides direct support for high-function display adapters being driven by GL applications.

load3dm5 The GAI 3D Model 5 load modules provide the rendering functions for supporting the OpenGL API. These load modules configure the rendering libraries depending on the amount of hardware support available for the API to use. This may range from full hardware support to full software support or some combination of the two. The 3D Model 5 was designed to fully support both direct (server bypass) and indirect (protocol to X server) rendering.

Kernel Components of the Display Subsystem

The kernel components are as follows:

LFT The LFT implements a simple character-oriented tty-like terminal emulation. LFT is not intended to be used in a steady state system environment. Customers are expected to use a graphical interface for all processing except system startup, installation and stand-alone diagnostics

RCM The rendering context manager (RCM) permits time sharing of the adapter between graphics processes. This sharing of the adapter is based upon system requests. It also permits space sharing of the adapter, in the sense of having the displayable area of the adapter. The RCM provides the environment for multiple graphics processes independently accessing display hardware. Each of these processes requests that the graphics adapter be in a particular state; the state is called the rendering context for that graphics process. It protects each graphics process from other graphics processes.

For 2D adapters only displaying X Window System clients, the RCM is not necessary.

Display Device Driver

This material covers configuration information that is unique to graphics and display device driver functions. Use it as a guide to the unique aspects of your adapter, the device driver you write, and how the X server and the LFT subsystem use your device.

Note: This section is preliminary and a prerequisite for writing the ddx portion of your adapter. It is recommended that this section be completed first before moving to the other sections.

LFT Overview

The LFT (low-function terminal) is a simple character oriented, tty-like terminal emulation. It replaces and simplifies the HFT (high-function terminal) of previous releases of AIX, no longer supporting virtual terminals or hotkeying. The LFT supports all graphics adapters and keyboards supported by AIX, but it does *not* support any other types of input devices such as mouse or tablet; these have their own drivers.

The LFT is configured during phase two of the boot process. The LFT will be configured only if there are graphics adapters already available. You must first configure your own adapter and then allow your device to be used by the LFT.

Configuration and ODM Object Classes

Most of the configuration information in the ODM related to devices tells the system management routines and other devices about your device. For graphics adapters some of the information is for graphics adapters only while other information is general about any device.

You must add the following objects for your device and information for other object classes so they can find your device. For example, the LFT and X server need to know if your device is available.

PdDv

The **PdDv** object class is the Predefined Device object class. It contains information about the device and the driver it uses, the configuration methods, class, subclass, and type information of the device.

PdAt

The **PdAt** object class contains information about various attributes of your driver that are unique to graphics. There are also **PdAt** attributes that are used by system management routines and the LFT subsystem.

Some of the attributes used for a graphics adapter are:

display_id	The **display_id** attribute takes a value of the form 0x04xx0000, where:
04	Fixed.
xx	An adapter specific ID. 0x00 – 0x7F are reserved forAIX use only. 0x80 – 0xFF are for vendor adapters. Use a value not in the database. Remember that all vendors use this range.
dsp_name	This is the value of type field of the **PdDv** object class. It provides a unique name for your adapter. Check the database to make sure your name is unique.
dsp_desc	Description of the adapter.
scrn_height	Screen height.
scrn_width	Screen width.

color1–16 The 16 default colors to be used by the VDD.

belongs_to Indicates that the LFT subsystem can use this device. The string **graphics** is the default value.

CuDep

The **CuDep** object class tells the LFT subsystem that your device is available for use. The LFT subsystem reads this class to find out which devices it can open and use as a graphics output device. You create an object in this object class based on the *belongs_to* attribute in the **PdAt** object class for that adapter.

GAI

The **GAI** object class is in the /**usr**/**lib**/**objrepos** directory. This object class requires two entries to tell the location of two modules that need to be loaded during X server startup. (Although these entries are not needed directly by the display driver to run the LFT, they are discussed here because all entries to the ODM should be handled by the define methods of the display driver.) The required entries are:

Adapter_Id This field is derived from the *display_id* attribute in the PdAt object class. This is the decimal equivalent of the *display_id* attribute value and is used as a link to the related objects in other object classes.

Module_Key This is the name of the module to be loaded when the X server starts.

Module_Path This is the path of the location of the module that is to be loaded. The X server prepends the string /**usr**/**lpp**/**gai**/ to this to form the full path name of the load module.

Configuration Summary

AIX configuration methods for a VDD (that is, the graphics adapter driver) do not create a /**dev** entry for their device. The device is exclusively used by the LFT subsystem. This simplifies supporting device multiplexing.

Instead of a user level /**dev** entry, the LFT subsystem calls kernel services such as **fp_opendev** and **fp_ioctl**. The LFT subsystem gets to the hardware specific functions of the VDD through the *d_dsdptr* pointer of the devsw table that it obtains through the **devswqry** kernel service.

VDD configuration methods should make sure that they save the address of the **phys_displays** structure in the *d_dsdptr* pointer of the devsw table. Initialize the **phys_displays** structure with pointers to various hardware specific functions in the VDD. If you decide to create a /**dev** entry for the adapter device, you need to incorporate privilege checks and protection mechanisms in your **open** and **close** routines.

The system finds out about your adapter through the **PdDv** object class and adds it to the system through your configuration methods. You allow your device to be used by the LFT through the *belongs_to* **PdAt** attribute.

Your configuration method puts an object into the **CuDep** object class that tells the LFT subsystem to use your graphics adapter. The system management commands **chdisp** and **lsdisp** use the name and description fields to show information about your adapter. Finally, the X server uses the *display_id* attribute to find out which ddx to load for your adapter. The information for which one to load is contained in the **GAI** object class.

Display Device Driver Subroutines

The standard display device driver subroutines are as follows:

- **vddconfig** routine
- **vddopen** routine
- **vddclose** routine
- **vddioctl** routine
- **interrupt handler**

Note: All subroutines with names beginning **vdd** (virtual device driver) deal with the low level display driver. Do not confuse these with routines with names beginning **vtt**, which are LFT Interface routines.

Configure the Device (vddconfig)

Purpose

Configures the display device driver.

Syntax

```
int vddconfig (devno, cmd, uiop)
dev_t devno;
int cmd;
struct uio *uiop;
```

Description

The **vddconfig** routine initializes the device driver into the device switch table. It also can terminate the device driver by removing itself from the device switch table.

The **vddconfig** routine is called by the display configure, unconfigure, or reconfigure method. This routine can also provide additional device specific functions relating to configuration such as returning device Vital Product Data. This routine is invoked through the **sysconfig** subroutine by the display configure method.

Parameters passed with this routine are:

devno Device major and minor number.

cmd What function this routine performs. The commands are:

INIT Specifies that the **vddconfig** routine is to perform an initialization function. This involves checking the minor number in *devno* for validity, and installing the device driver's entry points in the device switch table. This is accomplished by using the **devswadd** kernel service along with a **devsw** structure to add the device driver's entry points to the device switch table for the major device number supplied in the *devno* parameter.

This routine also copies the device dependant information from the Device Dependant Structure (DDS) provided by the caller into the device specific data area to be used when the device driver routines are invoked. The address and length of the DDS is described in the **uio** structure pointed to by the *uiop* parameter. The **uiomove** kernel service can copy the DDS into the data area of the device driver.

TERM Terminate the device driver and return any system resources. An unconfigure or reconfigure method, through the configuration entry point of the device driver, uses this command to remove resources and system access to this driver. This routine determines if any opens are outstanding on the specified *devno.* If not, it marks the device as terminated and does not allow any subsequent opens to the device.

All dynamically allocated data areas associated with the specified device number should be freed. If this termination removes the last minor number supported by the device driver from use, the **devswdel** kernel service should be called to remove the device driver's entry points from the device switch table for this *devno*.

Devices that can act as console should return an error without terminating the device driver.

QVPD Query VPD is an optional function used by the device's configure method to request the return of device specific vital product data. This information is usually used for diagnostic purposes. For this function, the **uio** structure pointed to by *uiop* must be set up by the caller to define an area in the caller's storage in which this routine is to write the vital product data. Use the **uiomove** kernel service to provide the data copy operation.

uiop A pointer to a **uio** structure specifying the location and length of the caller's data area in which to transfer information to or from.

This pointer is pointing to a caller provided **uio** structure that describes the location and length of the device-dependent data structure (for INIT) in which to read the information or to the vital product data area (for QVPD) in which to write the requested information. The **uiomove** kernel service can facilitate the copying of information out of or into the area described by the **uio** structure. The format of the uio structure is defined in the **sys**/**device.h** header file.

Return Values

The **vddconfig** routine should set the return code to zero if no errors were detected for the operation specified. If an error is returned, the return code is one of the values defined in the **errno.h** header file.

Open a Device (vddopen)

Purpose

Initializes the display device driver into the system.

Syntax

```
int vddopen (devno, devflag, chan, ext)
dev_t devno;
int devflag;
int chan;
int ext;
```

Description

The **vddopen** routine prepares a device for operation. The kernel calls **vddopen** when a program uses an **open** or **devopen** subroutine.

The display device driver can only be opened by one process at a time (LFT). The **open** subroutine can enforce this by maintaining a static flag variable, which is set to 1 if the device is open and zero if not. Each time it is called, **vddopen** checks the value of the flag and, if it is not zero, returns with a return code of EIO to indicate that the device is already open. Otherwise, **vddopen** sets the flag and returns normally. The **vddclose** routine later clears the flag when the device is closed.

The **vddopen** routine should initialize the device. It should allocate the required system resources to the device (such as DMA channels, interrupt levels and priorities) and register its **vddintr** device interrupt handler for the interrupt level required to support the target device (if required).

Parameters passed with this routine are:

devno Specifies both the major and minor device numbers.

devflag One of the following values:

DKERNEL The device was called by a kernel routine using DEVOPEN.

DREAD The device is being opened for reading only.

DWRITE The device is being opened for writing.

DAPPEND The device is being opened for appending.

DNDELAY The device is being opened in nonblock mode.

chan Channel number (ignored by this device driver).

ext The extended system call parameter (ignored by this device driver).

Return Values

The **vddopen** routine indicates an error condition to the application program by returning a nonzero return code. The return code should be one of the values defined in the **errno.h** header file.

Close a Device (vddclose)

Purpose

Resets the display device driver.

Syntax

```
int vdd_close (devno, chan, ext)
dev_t devno;
int chan;
int ext;
```

Description

The kernel calls the **vddclose** routine when a program uses a **close** or **devclose** subroutine.

The **vddclose** routine resets the display to prevent generating any more interrupts or DMA requests until it is opened again. It should free DMA channels and interrupt levels allocated for this device. The intent is to free system resources used by this device until they are needed again. The flag that was set by the **vddopen** routine should be reset.

Parameters passed with this routine are:

devno Specifies both the major and minor device numbers.

chan Channel number (ignored by this device driver).

ext The extended system call parameter (ignored by this device driver).

Return Values

The **vddclose** routine indicates an error condition by returning a nonzero return code. The return code should be one of the values defined in the **errno.h** header file.

Device Control (vddioctl)

Purpose

The **vddioctl** routine provides control commands and parameters to the device.

Syntax

```
long vdd_ioctl (devno, cmd, arg, devflag, chan, ext)
dev_t devno;
long cmd;
long arg;
ulong devflag;
long chan;
long ext;
```

Description

The **vddioctl** routine performs special I/O operations. These operations are normally device specific. Within the LFT subsystem it is not used, providing only access to adapter diagnostic functions. However, it is a valid porting strategy for upper level graphics libraries, such as the adapter ddx, to use **vddioctl**. Possible operations include returning adapter bus addresses for I/O, and DMA control.

Parameters passed with this routine are:

devno	Both the major and minor device numbers
cmd	Command parameter indicating what function this routine should perform
arg	Parameter from the **ioctl** subroutine call that specifies an additional argument for the *cmd* operation
devflag	Device open or other control flags
chan	Channel number (typically not used by this device driver)
ext	The extended system call parameter (typically not used by this device driver)

Return Values

The **vddioctl** routine indicates an error condition to the application program by returning a nonzero return code. The return code should be one of the values defined in the **errno.h** header file. Data may be returned to the user application by use of the **copyout** kernel service.

Programming Notes

There are *no* required ioctl commands. This entry point should always return success if there are no vendor-specific commands.

Interrupt Handling

Graphics adapters which generate interrupts need an interrupt handler. The interrupt handler is made known to the system during the device driver initialization. The specific interrupts to be handled are device dependent.

LFT Interface Routines

All the following routines take a pointer to the **vtmstruc** structure associated with the virtual terminal. The structure contains terminal specific data and pointers. The VDD uses this structure to obtain the pointer to its local terminal dependent data area, as well as to determine the position to move the cursor to when applicable. The following is a list of all the routines necessary to define the LFT interface:

- Activate (vttact)
- Copy full line (vttcfl)
- Clear rectangle (vttclr)
- Copy line (vttcpl)
- Deactivate (vttdact)
- Define cursor (vttdefc)
- Initialize (vttinit)
- Move cursor (vttmovc)
- Scroll display (vttscr)
- Terminater (vttterm)
- Draw text (vtttext)

Note: Structures are described in "Display Driver Structure Descriptions," on page 12-30.

Activate (vttact)

Synopsis

The **vttact** routine switches the Virtual Display Driver into the active state and gives the virtual display driver exclusive access to the display hardware.

Description

This routine copies the presentation space in the frame buffer and establishes the correct position of the hardware cursor. The presentation space is first cleared and then the cursor is placed in the upper left corner.

Also, the color palette is loaded.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttact)(vp);
struct vtmstruc *vp;
```

The parameter passed with this routine is:

vp Pointer to the **vtmstruc** structure associated with this terminal.

Return Values

The return value 0 indicates successful completion.

Programming Notes

Only one instance of the device driver can be in the activated state at one time.

Calling this routine when the device driver is already activated causes unpredictable effects on subsequent operations.

You *must* call the initialize (**vttinit**) routine (see page 12-24) prior to the first call to this routine. Failure to do so causes unpredictable effects on the operation of this and subsequent routines.

Copy Full Lines (vttcfl)

Synopsis

The **vttcfl** routine copies the entire or partial content of the presentation space by one or more full lines positions, either up or down. The source and destination points are within the presentation space and the resulting information which lies beyond the absolute lower right hand or upper left hand corner of the presentation space is lost.

Description

The command copies a sequence of one or more consecutive full lines of character and attribute pairs in either direction within a presentation space and truncates the modified content at the presentation space boundaries.

Use this command with the Clear Rectangle (**vttclr**) routine (see page 12-21) to insert new lines into the presentation space.

In addition, the cursor can optionally be removed or left shown on the screen.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttcfl) (vp, source_row, dest_row,

length, update_cursor);
struct vtmstruc *vp;
long source_row;
long dest_row;
long length;
ulong update_cursor;
```

Parameters passed with this routine are:

vp Pointer to the **vtmstruc** associated with this terminal.

source_row Row number at which to begin the line copy operation.

dest_row Row number to which the line copy operation will copy the first row of the source.

length Number of lines which are to be copied.

update_cursor
This is a Boolean value. If a device does not have a hardware cursor and this parameter is set to zero, the cursor is not visible when this command returns to the caller. Otherwise, the cursor is visible.

Note: If a display has a hardware sprite cursor, this routine does not affect the visibility of the cursor. Rather, UPDATE_CURSOR specifies whether the cursor is moved to the position specified in *vp–>mparms.cursor*.

Return Values

The return value 0 indicates successful completion.

Programming Notes

The actual amount of data copied varies, depending on the physical display adapter.

The cursor position used for positioning is contained in the **vtmstruc** structure pointed to by *vp*.

Clear Rectangle (vttclr)

Synopsis

The **vttclr** routine clears any specified rectangular area of the screen.

Description

This routine stores a space in each character position of the rectangular area with any combination of the following attributes:

- Foreground color (one of 16 different colors)
- Background color (one of 16 different colors)
- Font (one of 8 different fonts)
- Underscore
- Reverse image
- Blink
- Bright
- Non-display

If the real display does not have a hardware cursor, then the cursor may optionally be displayed or not displayed when this routine returns to the caller. If the real display has a hardware cursor, the cursor is always displayed when the routine returns to the caller.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttclr)(vp, screen_pos, attr, update_cursor);
struct vtmstruc *vp;
struc vtt_box_rc_parms *screen_pos;
ulong attr;
ulong update_cursor;
```

Parameters passed with this routine are:

vp Pointer to the **vtmstruc** structure associated with this terminal.

screen_pos Pointer to a **vtt_box_rc_parms** structure with the coordinates of the upper-left and lower-right corners of the rectangular area that is cleared. The **vtt_box_rc_parms** structure is defined in “Display Driver Structure Descriptions” on page 12-30.

attr Attributes of each character in the specified rectangular area.

update_cursor This is a Boolean value. If a device does not have a hardware cursor and this parameter is set to zero, the cursor is not visible when this command returns to the caller. Otherwise, the cursor is visible.

Note: If a display has a hardware sprite cursor, this command does not affect the visibility of the cursor. Rather, the *update_cursor* parameter specifies whether the cursor is moved to the position specified in the **vtmstruc** structure.

Return Values

The return value 0 indicates successful completion.

Programming Notes

If a rectangular area is not valid (such as coordinates outside the presentation space, upper-left and lower-right coordinates switch), the result of running this routine is unpredictable.

The cursor position used for positioning is in the **vtmstruc** structure pointed to by *vp*.

Copy Line Segment (vttcpl)

Synopsis

The **vttcpl** routine copies a specified segment in one or more consecutive lines either left or right.

Description

This routine copies a sequence of one or more consecutive character/attribute pairs in either direction within a line and truncates the modified content at the line boundaries.

The contents of the preceding lines are not affected. Succeeding consecutive lines, however, can be similarly copied if the number of lines requested to operate on is greater than one.

Use this routine with the Clear Rectangle (**vttclr**) routine (see page 12-21) to obtain the repetitive inline “move” function.

If the starting point is the first position in the first line and the number of lines requested is equal to the number of rows in the presentation space, then the entire screen is scrolled right. Otherwise, there is a partial screen scroll.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttcpl) (vp, rc, update_cursor);
struct vtmstruc  *vp;
struct vtt_rc_parms *rc;
ulong  update_cursor;
```

Parameters passed with this routine are:

vp Pointer to the **vtmstruc** structure associated with this terminal.

rc Pointer to a **vtt_rc_parms** structure containing the “copy to” and “copy from” information. This structure is defined in “Display Driver Structure Descriptions” on page 12-30.

string_length Number of character/attribute pairs to be copied in each line.

string_index Index of the first character to display.

start_row Beginning line on which to operate.

start_column Starting position, within each line, of the source to be copied from.

dest_row Last line on which to operate.

dest_column Starting position, within each line, of the destination to be copied to. If the destination column is bigger than the starting column, the specified line segment is copied to the right. Otherwise, it is copied to the left.

update_cursor This is a Boolean value. If a device does not have a hardware cursor and this parameter is set to zero, the cursor is not visible when this routine returns to the caller. Otherwise, the cursor is visible.

Note: If a display has a hardware sprite cursor, this routine does not affect the visibility of the cursor. Rather, *update_cursor* specifies whether the cursor is moved to the position specified in the **vtt_cursor** structure.

Return Values

The return value 0 indicates successful completion.

Programming Notes

The actual amount of data copied varies, dependent on the physical display adapter.

The cursor position used for positioning is contained in the **vtmstruc** structure pointed to by *vp*.

Deactivate (vttdact)

Synopsis

The vttdact routine switches the virtual display driver into the inactive state.

Description

Recall that the device driver model maintains a presentation space model. When the display hardware has a character buffer, that buffer will, in general, be used to contain and maintain the presentation space data while a given instance of the device driver is active (the so called integral presentation space case). If no hardware character buffer is used to maintain the presentation space the device driver must maintain a presentation space while either Active or Inactive.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttdact) (vp);
struct vtmstruc *vp;
```

The parameter passed with this routine is:

vp Pointer to the **vtmstruc** structure associated with this terminal.

Return Values

The return value 0 indicates successful completion.

Programming Notes

Calling this routine when the device driver is already deactivated causes unpredictable effects on subsequent operations.

Define Cursor (vttdefc)

Synopsis

The **vttdefc** routine changes the shape of the cursor to one of six predefined shapes.

Description

Select the shape of the cursor from a set of shapes which is dependent on the display device. Selectors 6 through 255 are reserved. The default (selector 2, the double underscore) is used if you specify a reserved selector.

This routine can also reposition and turn off the cursor.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttdefc) (vp, selector, update_cursor);
struct vtmstruc *vp;
uchar selector;
ulong update_cursor;
```

Parameters passed with this routine are:

vp Pointer to the **vtmstruc** structure associated with this terminal.

selector Cursor shape to display. The available shapes are:

0	No cursor
1	Single Underscorer
2	Double Underscore
3	Half Blob

4	Double Line
5	Full Blob
Other values	Default to double underscore.

update_cursor
This is a Boolean value. Whether the cursor should be made visible after its shape is changed.

Note: If a display has a hardware sprite cursor, this command does not affect the visibility of the cursor. Rather, Show_Cursor specifies whether the cursor is moved to the position specified in Cursor_Pos.

Return Values

The return value 0 indicates successful completion.

Programming Notes

The cursor is invisible on all displays if the selector value is zero (a null shape is chosen). The *update_cursor* parameter does not affect the visibility of the cursor on displays with hardware cursors.

If you specify an invalid cursor position, the results are unpredictable.

On display adapters that do not support a hardware cursor, the cursor shape is generated with an exclusive-or operation.

The cursor position used for positioning is contained in the **vtmstruc** structure pointed to by *vp*.

Initialize (vttinit)

Synopsis

The **vttinit** routine initializes the internal state of the virtual display driver. It also allows the caller to select up to eight different fonts for subsequent use by the virtual terminal (see the Draw Text (**vtttext**) subroutine on page 12-27). All characters in all the selected fonts must be the same size.

Description

This routine initializes the internal state of the virtual display driver. A list of eight font IDs may optionally be passed to this procedure.

All characters in all fonts in the list *must* be the same size. Any font whose character box size differs from that of the first font in the list is replaced by the first font in the list.

To allow custom font selection, vttinit is passed a font_id. Using the **chfont** command, the user can create a font_id which is stored in the ODM to be used at LFT configuration time. If font_id is not equal to –1, then the index is used to load the font table. If the font_id is equal to –1, then there has been no custom font selection, and the LFT uses the first available font in the font table.

If the font list is validated or a default font is found:

- The presentation space (PS) is initialized with space characters and the canonical attribute of each character in the PS is set to zero.
- The cursor is moved to the upper left corner of the PS and displayed if the virtual display driver is active.
- The default cursor shape (double underscore) is selected.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttinit)(vp, font_ids, ps_size);
struct vtmstruc *vp;
struct fontpal  *font_ids;
struct ps_s *ps_size;
```

Parameters passed with this routine are:

vp Pointer to the virtual-terminal-specific data area.

font_ids Pointer to a **fontpal** structure that contains the font IDs to use for the various fonts selected by the attribute encoding. The font order is important as it matches the font selector enumerated scalar in the canonical representation of attributes (see “vtt_cp_parms” on page 12-30). This parameter is optional. If it is not specified, one default font will be selected.

ps_size Pointer to a **ps_s** structure that contains the width and height (in characters) of the presentation space.

Return Values

This routine sets the height and width of the presentation space in the *ps_s* parameter. If it is unable to allocate the local data, vttinit returns ENOMEM. The return value 0 indicates successful completion.

Programming Notes

All fonts specified in this routine should be the *same size* because of the default actions.

Call this routine *before* the first call to the activate (**vttact**) routine (see page 12-19). If you do not, there are unpredictable effects on subsequent operations.

The PS size is calculated using the following formulas (all division is integer division):

$$\textbf{Presentation Space Height} = \frac{\textbf{Height of the real screen (in picture elements)}}{\textbf{Number of rows in a character box}}$$

$$\textbf{Presentation Space Width} = \frac{\textbf{Width of the real screen (in picture elements)}}{\textbf{Number of columns in a character box}}$$

Move Cursor (vttmovc)

Synopsis

The **vttmovc** routine moves the cursor to the indicated position.

Description

The current cursor shape is repositioned to the specified character row and column. The cursor is always visible after this routine if the cursor has been defined as a visible shape, and this virtual terminal is active.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttmovc) (vp);
struct vtmstruc *vp;
```

The parameter passed with this routine is:

vp Pointer to the **vtmstruc** structure associated with this terminal.

Return Values

The return value 0 indicates successful completion.

Programming Notes

If an invalid position is specified, the results are unpredictable.

Scroll (vttscr)

Synopsis

The **vttscr** routine scrolls the entire contents of the display screen up or down.

Description

The current display screen contents are moved the indicated number of lines up or down. When scrolling up, the lines moved off the screen at the top are discarded and the indicated number of lines at the bottom are cleared to blanks with the attributes provided. When scrolling down, the lines moved off the screen at the bottom are discarded and the indicated number of lines at the top are cleared to blanks with the attributes provided. The cursor may also be repositioned and made invisible.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttscr)(vp, lines, attributes,
update_cursor);
struct vtmstruc *vp;
long lines;
ulong attributes;
ulong update_cursor;
```

Parameters passed with this routine are:

vp Pointer to the **vtmstruc** structure associated with this terminal.

lines Number of lines to scroll. A positive value moves the screen contents toward the top. A negative value moves the screen contents toward the bottom of the screen.

attributes Attributes of the blanks to be inserted in the new lines that appear at either the top or bottom of the screen.

update_cursor
This is a Boolean value. Whether the cursor should be made visible after the scroll.

Note: If a display has a hardware sprite cursor, this command does not affect the visibility of the cursor. Rather, Show_Cursor specifies whether the cursor is moved to the position specified in Cursor_Pos.

Return Values

There are no return values.

Programming Notes

If a position is not valid, the results are unpredictable.

The cursor position used for positioning is contained in the **vtmstruc** structure pointed to by *vp*.

Terminate (vttterm)

Synopsis

The **vttterm** routine prevents interrupts to a virtual terminal and must be issued when the virtual terminal closes.

Description

A virtual terminal in the process of closing no longer requires interrupts from its I/O devices. In fact, after the virtual terminal process terminates, virtual interrupts directed to that process cause a system failure. To prevent such problems, the virtual terminal must call the terminate routine before terminating itself.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vttterm)(vp);
struct vtmstruc *vp;
```

The parameter passed with this routine is:

vp Pointer to the **vtmstruc** structure associated with this terminal.

Return Values

The return value 0 indicates successful completion.

Programming Notes

Not issuing this routine when closing causes a system failure.

For display adapters that generate interrupts serviced by the virtual terminal, the routine ensures that the adapter is not reinitialized when the virtual terminal process terminates.

For device drivers which allocate storage for their presentation space, this routine will free that storage and all the local data areas.

Draw Text (vtttext)

Synopsis

The **vtttext** routine draws a string of qualified ASCII characters into the refresh buffer and presentation space buffer of the display device.

Description

Each supplied ASCII character is drawn into the refresh buffer and/or presentation space buffer beginning at the specified starting row and column. At the end of this character drawing operation, the cursor is redrawn at the specified new location.

Each character drawn is mapped to the appropriate font range by two mechanisms. First, each character will be logically ANDed by the value supplied in the code point mask parameter. It is expected that this parameter will contain either 0xFF or 0x7F which will have the effect of wrapping the code point in either a seven-bit range or an eight-bit range. Next, the code point base parameter is added to each supplied character to find the correct symbol in the font.

Each character will be drawn with a device specific attribute derived from the canonical attribute specification in the *vtt_attr* parameter.

The cursor can be moved or redrawn by this routine if the *update_cursor* parameter is set to true (1). In this case, a cursor of the type specified by the Define Cursor (**vttdefc**) routine (see page 12-23) will be moved or redrawn, as appropriate for the given hardware at the location specified in the **vtmstruc** parameters.

Note: The no show state (zero) of *update_cursor* does not guarantee that the cursor will be invisible after this operation. Whether or not it is invisible is a device-dependent characteristic.

Recall that the device driver model maintains a presentation space model. Usually, where the display hardware has a character buffer, that buffer is used to contain and maintain the presentation space data while a given instance of the device driver is active. This is the *integral presentation space case*.

If no hardware character buffer is used to maintain the presentation space, this is the *disjoint refresh buffer case*, in which the driver must maintain an off adapter presentation space while active or inactive.

Syntax

Use this routine with the following call:

```
rc = (*vp->display->vtttext)(vp, ascii_string, rc, cp,
update_cursor);
struct vtmstruc *vp;
char *ascii_string;
struct vtt_rc_parms *rc;
struct vtt_cp_parms *cp;
ulong update_cursor;
```

Parameters passed with this routine are:

vp Pointer to the **vtmstruc** structure associated with this terminal.

ascii_string Adjustable extent array of characters which are to be drawn on the display. The length of this string must be greater than or equal to the *string_length* parameter in the **vtt_rc_parms** structure.

rc Pointer to a **vtt_rc_parms** structure containing the ASCII string, row, and column information.

string_length How many of the characters in the ASCII string are to be drawn. Note that this drawing operation performs no end-of-line check so that unpredictable results will occur if the combination of *string_length* and *start_column* causes an attempt to write off the end of the line.

string_index Index of the first character in *ascii_string* that is to be displayed.

start_row Specifies, in the presentation buffer and display refresh buffer, the row number of the first character drawn. This parameter has a unity origin.

start_column Specifies, in the presentation buffer and display refresh buffer, the column number of the first character drawn. This parameter has a unity origin.

dest_row
dest_column Not used.

cp Pointer to a **vtt_cp_parms** structure containing the ASCII string, and row and column information.

cp_mask 8-bit value which is logically ANDed with each ASCII character before font translation. Legal values for this parameter are either 0xFF or 0x7F.

cp_base Value which allows each ASCII character to be translated to a final font set, larger than 256 code points. The ASCII character is masked by *cp_mask*. The masked ASCII character is logically added to the value of *cp_base* to determine the actual character, in a 10-bit selection, to be displayed.

attributes Canonical representation of the desired attributes which should be used when drawing the character string. See the discussion of canonical attribute representation under “vtt_cp_parms” on page 12-30.

cursor A **vtt_cursor** structure that defines the x and y position of the cursor.

update_cursor

This parameter is a Boolean value and controls whether the cursor should be redrawn at the end of this draw routine. The purpose is to allow the LFT subsystem to control how much APA display processing is wasted in undrawing and redrawing cursors.

Note: If a display has a hardware sprite cursor, this routine does not affect the visibility of the cursor. Rather, *update_cursor* specifies whether or not the cursor is moved to the position specified in the **vtt_cp_parms** structure.

Return Values

There are no return values.

Programming Notes

Use this routine *only* to draw in one *single* line of the display at a time. Length specifications that would imply a wrap to next line in the middle of the call will cause unpredictable results in displays with invisible refresh buffer space at the end of the visible line.

The cursor position used for positioning is contained in the **vtmstruc** structure pointed to by *vp*.

Display Driver Structure Descriptions

vtt_rc_parms

The **vtt_rc_parms** structure is supplied as a parameter to several of the virtual display driver routines, notably the Draw Text routine, the two copy routines and the Read Screen Segment routine.

```
/*****************************************************************/
/* row column length structure interfaces to VDD Routine         */
/*****************************************************************/
struct vtt_rc_parms {
        ulong string_length;    /* length of character string that    */
                                /*   must be displayed                */
        ulong string_index;     /* index of the 1st char to display   */
        long  start_row;        /* starting row for draw/move         */
                                /*   operations, unity based.         */
        long  start_column;     /* starting column for draw/move      */
                                /*   operations, unity based          */
        long  dest_row;         /* destination row number for move    */
                                /*   operations, zero based           */
        long  dest_column;      /* destination column number for move */
                                /*  operations, zero based            */
};
```

vtt_box_rc_parms

The **vtt_box_rc_parms** structure is supplied as a parameter to the Clear Rectangle routine entry points.

```
/*****************************************************************/
/* the vtt_box_rc_parms structure is supplied as a parameter to the */
/* VDD clear rectangle routine.                                  */
/*****************************************************************/
struct vtt_box_rc_parms {
        long row_ul;            /* row number of upper-left corner   */
                                /*   of the upright rectangle        */
        long column_ul;         /* col number of upper-left corner   */
                                /*   of the upright rectangle        */
        long row_lr;            /* row number of lower-right corner  */
                                /*   of the uproght rectangle.       */
        long column_lr;         /* col number of lower-right corner  */
                                /*   of the  upright rectangle       */
};
```

vtt_cp_parms

The **vtt_cp_parms** structure is supplied as a parameter to the Draw Text routine. The **vtt_cursor** substructure is passed as a parameter to several procedures such as Define Cursor, Move Cursor, and Scroll Up.

```
/*****************************************************************/
/* code point base/mask and cursor positioning parameter      */
/* structure for use in vtttext  replace text                 */
/*****************************************************************/
struct vtt_cp_parms
{
        ulong cp_mask;      /* code point mask for implementing */
                            /*   7 or 8 bit ascii               */
        long cp_base;       /* code point base, added to code   */
                            /*   point if base >= 0             */
        ushort attributes;  /* attribute bits                   */
        struct  vtt_cursor cursor;  /* cursor x and y position  */
};
```

The definitions and default states of the canonical attributes are:

FG COLOR Foreground color specification in writing into the frame buffer. The default value is device dependent.

BG COLOR Background color specification to be used in writing into the frame buffer. The default value is device dependent.

FONT_SELECT
Select for the font to be used in writing into the frame buffer. A value of zero selects the first font ID supplied in the VTTINIT font ID array as the active font for drawing. A value of seven selects the eighth font ID array value as the active font for drawing. The default value is zero.

NO_DISP Select non-displayed mode for characters. One equals non-displayed, zero equals displayed. Default value is zero.

BRIGHT Select bright (intensified) display mode. One equals intensified. Zero equals normal intensity. Default value is zero.

BLINK Select blinking representation. One equals blinking characters. Zero equals non-blink. Default value is zero.

REV_VIDEO Select reverse video representation. One equals reversed image. Zero equals normal image. Default value is zero. To get the effect of reverse video on all adapters, this bit must be set to one. Swapping the foreground and background colors will not yield the desired result on all adapters because these fields may in fact be ignored by some virtual device drivers.

UNDERSCORE
Select underscoring of characters. One equals underscore each character. Zero equals no underscore. Default value is zero. A default color table is provided for any display type. The size and organization of color tables is device dependent.

No single device supports all the canonical attributes in any configuration. For example, when a canonical attribute is supplied to the Draw Text routine, the canonical form is interpreted to something usable by the given device. When a subsequent call to the Read Screen Segment routine is made, the internally stored attributes will be mapped back to canonical form. This back transform has some loss of information in that canonical attributes that are unsupported by a given device will be set to a default state instead of the originally supplied state.

font_data

The **font_data** structure is initialized by the LFT and is used by the VDD to get the necessary font information for displaying text.

```
struct font_data
{
        ulong font_id;          /* font_data index (1 based) */
        ulong font_attr;        /* 0=plain, 1=bold, 2=italic */
        ulong font_style;       /* 0=apa,*/
        long  font_width;       /* charbox width  in pels*/
        long  font_height;      /* charbox height in pels*/
        long  *font_ptr;        /* pointer to font file*/
        ulong font_size;        /* size of font structure*/
};
```

phys_displays

The **phys_displays** structure is defined in **/usr/include/sys/display.h**.

```
/*******************************************************/
/* presentation space structure                        */
/*******************************************************/

struct ps_s {
        long ps_w;        /* presentation space width  */
        long ps_h;        /* presentation space height */
};

/*************************************************************/
/* cursor positioning structure used as parameter to VDD routine */
/*************************************************************/

struct vtt_cursor {
         long  x;
         long  y;
};

struct vtmstruc {
       struct phys_displays *displays;     /* display this VT is using */
       struct vtt_cp_parms  mparms;        /* attribute+cursor position*/
       char                 *vttld;        /* pointer to VTT local data*/
       off_t                vtid;          /*virtual terminal id = 0   */
       uchar                vtm_mode;      /* mode = KSR               */
       int                  font_index;    /* -1 means use 'best' font */
       int                  number_of_fonts; /* number of fonts found  */
       struct font_data     *fonts;        /* font information         */
       int                  (*fsp_enq)();  /* font request enqueue func*/
};
```

Device Dependent Structure (DDS)

The Device Dependent Structure is specific to the display adapter. The contents must contain the addresses, interrupt levels, DMA areas, and other configuration information set by the configuration method.

The following is an example Device Dependent Structure:

```
struct display_dds
{
         int adapter_busaddr;
         int int_level;
         int int_class;
         int slot_number;
         int color_tablet[16];
         int dms_channel;
         int dma_address;
         int screen_height_mm;
         int screen_weight_mm;
         int display_id;
};
```

Graphics Adapter Interface (GAI) 2D Adapter Load Modules

The device-dependent portion (ddx) of the X server code (the portion of the server code that actually manipulates the adapter) is located in dynamically loadable modules under the current GAI architecture. These modules are loaded at server invocation time when a user requests that the server run on a specific adapter via command-line flags, or the default adapter(s) in the absence of a specific request. Adapter modules not requested in this instantiation of the server are not loaded.

The interface between the device-independent (dix) portion of the server and the device dependent (ddx) subsystem is documented in *Strategies for Porting the X11 Sample Server* by Angebranndt and others. The porting layer specified in the porting guide is a well-defined interface which allows third party vendors to supply those device-dependent portions of the X server required to support a new device. In the current GAI architecture on AIX, the ddx resides in dynamically loadable, device-dependent code modules (loadddx's) and are required to support execution of the X server on display adapters. By utilizing a dynamic loading scheme, the GAI architecture facilitates the independent addition of new driver modules by third-party vendors, as access to object modules for static binding is not necessary.

The loadable module technique is utilized to support the addition of extensions to the X Window System as well. Loading only the required extensions for a particular invocation of the server limits the system resources from being consumed. An additional benefit is that third-party extension writers can add their X server extensions without requiring server integration and rebuilding.

This information highlights the process of initializing and destroying 2D minimal RMS adapter instances within the GAI architecture. Details regarding the implementation of an X device dependent subsystem (ddx) can be found in standard documents, such as *The X Window System Server* by Israel and Fortune.

Loadable DDX Interface

The process by which the X server and one or more adapters is initialized is divided into three steps:

- Selecting the adapter or adapters through command-line flags or default actions
- Loading the appropriate load module and dynamic binding with X server executable
- Initializing the hardware and the X server screen structure

Selection of Adapters

Command-line flags allow the user to specify a number of characteristics of the X server display. Characteristics relating to adapter selection include the number of screens for the display, configuration of the screens for the display, and default visual and default depth of the display. Command-line flags for adapters include the logical name returned from the AIX **lsdisp** command as they are stored in the Object Data Manager database (the ODM database contains the configuration information for the entire system). As an example, the **lsdisp** command returns the following information if the system is configured with a Color Graphics Display Adapter and a POWER_Gt1 Graphics Adapter:

```
DEV_NAME        SLOT    BUS     ADPT_NAME       DESCRIPTION

sga0            0J      sys     POWER_Gt1       Graphics Adapter
gda0            03      mca     colorgda        Graphics Adapter
```

If no display characteristic command-line flags are specified, the default actions are used. These default actions include:

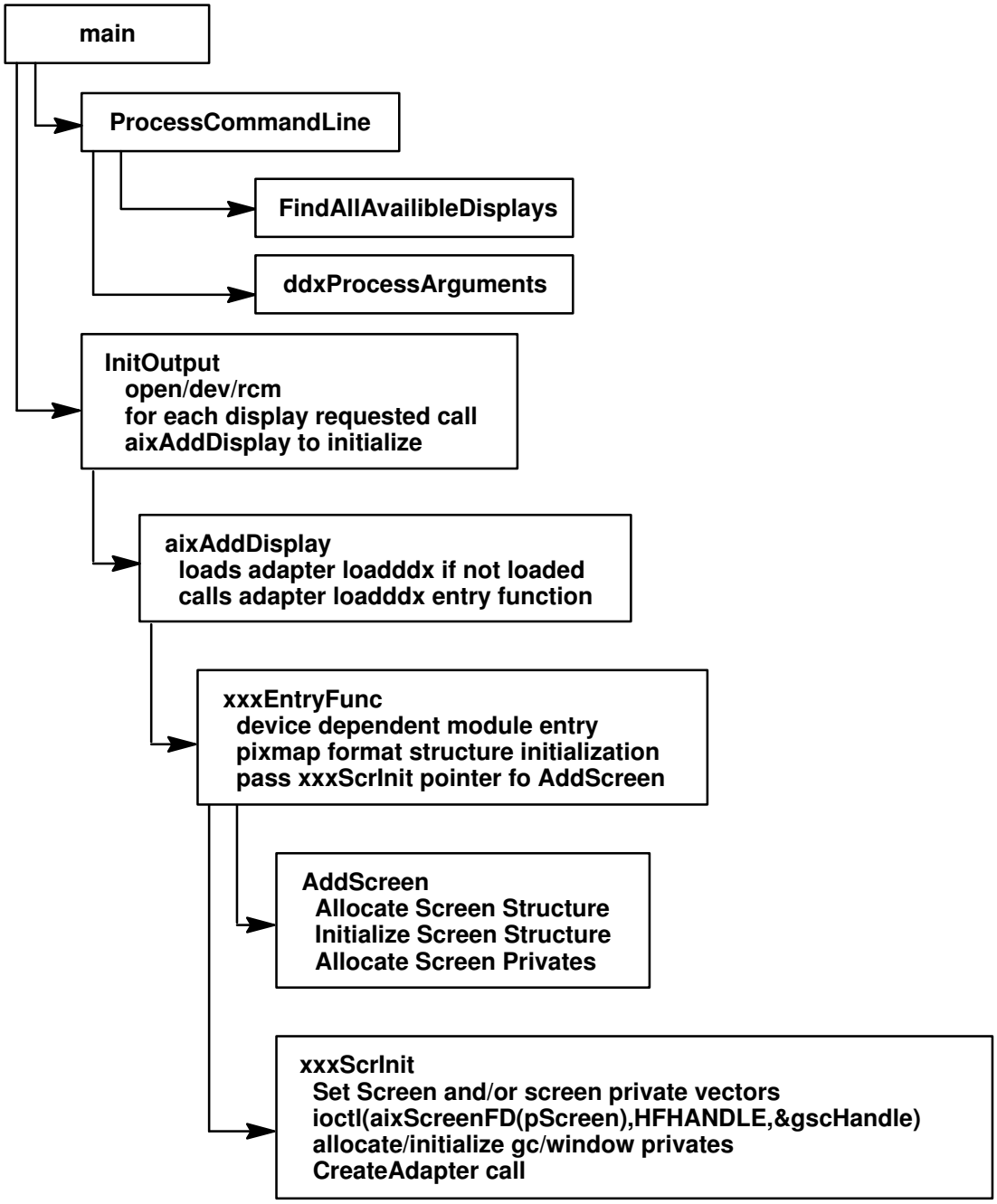
- Use of all configured adapters. If there is only one graphics adapter installed, the X server will be initialized on that graphics adapter. If more than one adapter is configured in the system, the X server will be initialized in multihead mode.

- Use of PseudoColor visual for the default visual (or GrayScale visual for grayscale adapters).
- Use of default depth (usually 8).

If the user chooses to bypass the default server configuration options and initialize on less than all available display adapters, the –P *Row Column DeviceName* command-line option must be specified. If the user chose to initialize the X server on Color Graphics Display Adapter (assuming output from **lsdisp** above), the appropriate command line options would include the flag to initialize on Color GDA only, `-P11 gda0`.

As part of the adapter initialization process, the X server main function calls an operating system dependent subroutine, **ProcessCommandLine**. **ProcessCommandLine** calls **FindAllAvailableDisplays** function, which queries the ODM database for all available graphics adapters in the system and sets up a **DisplayRec** structure for every adapter found. The **DisplayRec** structures are actually stored in a global dynamic array, **AvailableDisplays**. **FindAllAvailableDisplays** returns to **ProcessCommandLine** which calls a device-dependent command line parsing function, **ddxProcessArgument**. **ddxProcessArgument** handles some GAI-specific global variable initializations based on the command line flags specified for wrapping pointer in x and y directions, backing-store configuration, extension loading, and adapter initialization. It is in **ddxProcessArgument** that the remaining structure members in the adapter **DisplayRecs** are initialized to specify users interest in X initialization on a given adapter. When **ddxProcessArgument** encounters a –P flag, it uses the logical name associated with the flag as a key in the ODM **CuDv** database and retrieves the associated entry. If the entry does not exist, an error message is displayed. When an unexpected option or invalid value is specified for a flag, the X server displays the usage message and ignores the option or terminates initialization, depending on the severity of usage error. Both **ProcessCommandLine** and **ddxProcessArgument** validate command line options and display error messages.

Assuming that there are no errors detected by **ddxProcessArgument**, **InitOutput** is called from the **main** subroutine, passing a pointer to the **screenInfo** structure. The pixmap and bitmap format information is initialized in the **screenInfo** structure and for each requested adapter **aixAddDisplay** is called to load the loadddx if it has not been bound in (on server reset, the module would have already been loaded) and evoke the drivers entry function, for example **xxxEntryFunc** which sets device specific **pixmapFormat** information and calls the MIT dix function **AddScreen** which “increases” the size of the **screenRecs** (internal server structure) array and calls the device-specific initialization routine specifying the adapters screen number, a pointer to its **ScreenRec**, and the command line argument vector and argument count where the adapter will be initialized, screen vectors set, and access to the adapter given to X and rms configured via **CreateAdapter** call.



Server Initialization Sequence for the Selection and Initialization of Displays

X Server Initialization Subroutines

Note: The following subroutines are for informational purposes only.

ddxProcessArgument

Purpose

Processes the command line arguments for the X server.

Syntax

int ddxProcessArgument (*argc***,** *argv***,** *i***)**
int *argc***;**
char ****argv*[];**
int *i***;**

Description

The **ddxProcessArgument** subroutine is an AIXwindows supplied subroutine. There is one for the X server rather than one for each ddx. It is called for each flag one flag at a time. Its parameters include the *argc* and *argv* parameters which were used for this invocation for the X server and the current index into *argv*. **ddxProcessArgument** looks at the current argument and returns zero if the argument is not a device-dependent one, and otherwise returns a count of the number of elements of *argv* that are part of this one argument.

Parameters

argc	Specifies the number of options in the *argv* list.
argv	Specifies an array of options that were passed to the X Server at its invocation.
i	Specifies the current index into *argv* list.

Flags

ddxProcessArgument processes the following arguments (not a complete list):

Mouse Processing Flags

Flag Name	Variable Set
–wrapx	**aixXWrapScreen** Specifies mouse behavior when the mouse's hot spot reaches the left or right borders of any root window.
–wrapy	**aixYWrapScreen** Specificies of mouse behavior when the mouse's hot spot reaches the top or bottom borders of any root window.
–wrap	**aixXWrapScreen, aixYWrapScreen** Sets both the **–wrapx** and **–wrapy** flags.

Other Flags

Flag Name	Variable Set
–bs	**aixAllowBackingStore** Enables backing store support on all screens.
–nobs	**aixAllowBackingStore** Disables backing store support on all screens.
–x <extention name>	**how_many_extensions, extension_list** Tracks number of and list of requested extensions.

–T **aixDontZap**
Disables the Ctrl-Alt-Backspace key sequence used to kill the server**.**

–wp **aixWhitePixelText**
Allows you to specify a WhitePixel color.

–bp **aixBlackPixelText**
Allows you to specify a BlackPixel color.

–n **dpy**
Specifies the connection number of the server**.**

–layer <int> layer
Specifies the requested layer for the default visual if the ddx supports overlays and underlays.

Processing of Displays Specified on the Command Line

The following flags can be used to specify the display on the command line. If a display is not specified, by default, the X server is invoked on all available displays in a multihead configuration by slot order. The first active display found becomes the equivalent of `-P11` (Screen 0). The next active screen found becomes the equivalent of `-P12` (Screen1).

–force Specifies that the X server may be invoked from tty.

–P *Row Col DisplayName*
Specifies that the X server is to be invoked on the display specified by the display_name parameter. Valid values for this field are those found in the logical_name field of the **CuDv**.

In all of the flags, the *display_name* parameter is specified by the logical name of the device. You can find out what this logical name is by executing the **lsdisp** command.

Note: Only the **ProcessCommandLine** and **ddxProcessArgument** subroutines should process command-line flags.

FindAllAvailableDisplays

Purpose

Initializes the **DisplayRec** structure for the X server.

Syntax

void FindAllAvailableDisplays (*argc***,** *argv***)**
int *argc***;**
char *****argv*;**

Description

The **FindAllAvailableDisplays** subroutine queries the ODM to find out information about all possible graphics displays that are available. It returns a initialized **DisplayRec** structure for each display and sets the **NumAvailableDisplays** global variable to the number of displays attached to the system.

The information needed to obtain a list of all available displays is contained in two ODM classes, the **PdAt** (predefined attributes) and the **CuDv** (customized devices). To find all the active displays attached to a system, two ODM queries are needed—one to find all the graphics devices, and one to subset the available devices from all the graphics devices.

For each display attached to the hardware, the **FindAllDisplays** subroutine initializes the following information:

dev_id Specifies the GAI adapter ID. This is used as a key to the ODM query of the GAI load modules. This value is stored in the predefined attribute **display_id**.

name Set to the name of the display as returned by the **lsdisp** command (i.e. ppr0). This value is the logical name of the device as retrieved from the PdAt query of all the graphics devices.

loadModule Set to the loadddx load module for the specified adapter. It is a concatenation of the Module_path as stored in the GAI class and the **GAI_PATH**. The value stored is the full path not the partial path as retrieved from the GAI.

EntryFunc Set to **NULL**. This gets initialized when the loadddx load module is actually loaded.

x_pos Set to 0. This is initialized to the appropriate value later in the initialization cycle.

y_pos Set to 0. This is initialized to the appropriate value later in the initialization cycle.

requested Set to **False**. Set to True later in the initialization cycle when the adapter is actually requested.

display_number
Set to the current value of the variable used to control the for loop.

Parameters

The *argc* and *argv* (count and vector) command-line arguments are passes in for processing of the **–force** flag. The **–force** flag allows the X server to be started from a tty.

InitOutput Subroutine

Purpose

Initializes the **screenInfo** *imageByteOrder*, bitmap information, and pixmap format information.

Syntax

void InitOutput (*screenInfoPtr***,** *argc***,** *argv***)**
ScreenInfo ****screenInfoPtr*;**
int *argc***;**
char ****argv*[];**

Description

The **InitOutput** subroutine initializes the **screenInfo** *imageByteOrder*, bitmap information, and pixmap format information. The **InitOutput** subroutine obtains a private index for the screen and one for the AIX extensions. These indexes are global variables named **aixExtScreenPrivateIndex** and **aixScreenPrivateIndex.** They are used as private indexes for the structure containing the vectors for the AIX private extensions (cursor and colormap) and the AIX screen private structure and vectors.

If there were no screens specified on the command line, the subroutine processes the default screens. The default screens are all currently available screens, the first screen found in the ODM.

The **InitOutput** subroutine opens the /**dev**/**rcm** file once for each invocation of the X server and not done once per head in a multihead environment. The subroutine stores the file descriptor from the open in the **screenInfoPriv** structure.

For each screen, the **InitOutput** subroutine calls subroutines to load and initialize or reinitialize the adapter. The **serverGeneration** global variable can be used to determine if the X server is being invoked or reset.

Parameters

screenInfoPtr	Specifies a pointer to the **screenInfo** structure. The **screenInfo** structure contains a section of per server information as well as an array of screen information.
argc	Specifies the number of options in the *argv* list.
argv	Specifies an array of options that were passed to the X server at its invocation.

Device-Dependent Initialization Subroutines

Note: These routines must be provided by the **loadddx** module.

xxxentryFunc Subroutine

Purpose

Returns information from the loading of the **loadddx** module.

Syntax

int xxxentryFunc (*index***,** *pScreen***,** *argc***,** *argv***)**
register int *index***;**
register ScreenPtr *pScreen***;**
int *argc***;**
char *****argv*;**

Description

The address of the entry point of the loadable driver is returned when the **load** system call is performed on the **loadddx** module. The function is invoked in the **InitOutput** subroutine and performs some or all of several actions. This function must be provided by the load module driver.

The **xxxentryFunc** function initializes any appropriate pixmap formats and then calls **AddScreen** subroutine to allocate the per screen structure and provide addition dix initialization of the **screenInfo** structure. The **AddScreen** subroutine allocates the screen structure, links it appropriately into dix structures and then calls a ddx supplied routine to finish the initialization.

A pointer to **xxxentryFunc** is stored in the *EntryFunc* field of the **DisplayRec** structure for this adapter. In practice, this function can be called whatever the driver implementor chooses. By convention it is referred to as **xxxentryFunc**.

Parameters

index	Specifies the index of this screen in the array of screens for the X server.
pScreen	Specifies a pointer to the screen structure.
argc	Specifies the number of options in the *argv* list.
argv	Specifies an array of options that were passed to the X server at its invocation.

Return Values

Success

Failure

xxxScrInit

Purpose

Initializes the screen structure for each screen and performs device dependent initialization as required.

Syntax

Bool xxxScrInit (*index***, *pScreen*, *argc*, *argv*)**
register int *index*;
register ScreenPtr *pScreen*;
int *argc*;
char ***argv*;

Description

There is one **xxxScrInit** subroutine for each adapter load module. Its address is passed to the **AddScreen** dix subroutine and is called from the **AddScreen** subroutine through the function pointer to do device dependent initialization of the per-screen structure as well as window and gc devPrivates. The **xxx** in the subroutine name is replaced with some meaningful prefix, such as the adapter name. This subroutine performs the following:

- Gets a **gscHandle** from the RCM by making an ioctl call with a GSC_HANDLE parameter.
- Calls the **CreateAdapter** function for the 2D model.
- Registers this process as a graphics process by making the **aixgsc** system call with the MAKE_GP parameter.
- Initializes **pScreen** structure. This includes any visual structures for the screen.
- Allocates and initializes **pScreenPrivate** vectors (calls **aixAllocatePrivates**) and initializes AIX extension vectors—these are the vectors for the cursor and colormap extensions.
- Allocates and initializes any internally used private structures.
- Performs any other device-specific initialization.

Parameters

index	Specifies the index of this screen in the array of screens for the X Server.
pScreen	Specifies a pointer to the screen structure.
argc	Specifies the number of options in the *argv* list.
argv	Specifies an array of options that were passed to the X Server at its invocation.

Return Values

True	**xxxScrInit** succeeded
False	**xxxScrInit** failed.

xxxCloseScreen

Purpose

Closes the screen during X Server reset.

Syntax

Bool pScreen–>CloseScreen (*scrnNum***, *pScreen*, *argc*, *argv*)**
int *scrnNum*;
ScreenPtr *pScreen*;
int *argc*;
char ***argv*;

Description

The **xxxCloseScreen** subroutine is a device-dependent routine that must be provided by the driver implementor. This subroutine is called for each initialized screen when the server is reset or closed down.

The **xxxCloseScreen** subroutine should free any privately allocated structures. If dynamically allocated space is not freed, the driver will cause the X server to leak memory across server resets. In addition, a process similar to that outlined in the **xxxScrInit** subroutine must be followed to relinquish access to the adapter.

Whereas in the **xxxScrinit** routine the MAKE_GP subfunction of the aixgsc system call was called to gain access to the adapter, the UNMAKE_GP subfunction of **aixgsc** must be called in the **xxxCloseScreen** routine to relinquish access to the adapter. Failure to do so will prevent the success of subsequent **aixgsc** MAKE_GP calls in **xxxScrInit** if the server is resetting and not actually terminating.

Parameters

scrnNum	Number of screen
pScreen	Specifies the screen to be closed.
argc	Number of command line parameters
argv	Command line argument vector

Server Termination

Each implementation of the X server must provide two routines that terminate the X server. The **ddxGiveUp** subroutine is called when the X server terminates normally. The **abortDDX** subroutine is called when the X server encounters an unrecoverable error. Both of these subroutines close the screen for each active screen of the display. There is no change in either subroutine.

Adapter Access and the aixgsc System Call

Because the X server is a user process, special considerations have to be taken to ensure that the DDX is able to access the adapter. This is done through the **aixgsc** system call with the MAKE_GP parameter.

The MAKE_GP parameter marks the calling process as a graphics process for the specified adapter and returns a segment base address of the adapter and a device-specific structure that should contain device-specific offset addresses. The device-specific structure may also contain additional information.

The device-specific portion of the information returned by the **aixgsc** system call is actually set in the low-level device driver **make_gp** routine. This is a hardware-specific function that is a member of the **phys_display** structure. (See “Display Device Driver” on page 12-13.) This make_gp routine fills in addresses and other information required by the DDX.

Once the **make_gp** routine of the display device driver has been properly implemented, the **aixgsc** system call will return the correct addresses within the loadable DDX module. This enables the X server to access the graphics adapter.

Implementation Details

The **aixgsc** system call takes a pointer to the following structure (defined in **/usr/sys/include/aixgsc.h**) as an argument:

```
typedef struct _make_gp  {
    int        error;        /* error report */
    caddr_t    segment;      /* segment base address */
    genericPtr pData;        /* device specific adapter addresses
*/
    int        length;       /* length of device specific data */
    int        access;       /* access authority */
#   define SHARE_ACCESS 0     /* process shares access to adapter
*/
#   define EXCLUSIVE_ACCESS 1 /* process cannot share access */
} make_gp;
```

The following is a sample **make_gp** routine for a display device driver:

```
static long xxx_make_gp(pdev,pproc,map,len,trace)
gscDev *pdev;
rcmProcPtr pproc;
struct xxx_map *map
int len;
int *trace;
{
   struct ddsent * dds;

   /* This routine fills in the register mappings for this instance */
   /* returns to the gsc call.                                      */

   dds = (struct ddsent *) pdev->devHead.display->odmdds;
   map->io_addr = dds->io_bus_addr_start;
   map->cp_addr = dds->io_bus_mem_start;
   map->vr_addr = dds->vram_start;
   map->dma_addr[0] = dds->dma_range_start;
   map->screen_height_mm = dds->screen_height_mm;
   map->screen_wdith_mm = dds->screen_width_mm;
   return(0);
}
```

With the above **make_gp** routine, the following is a typical fragment of code to bind the graphics adapter load module into the GAI architecture and allow access to the adapter within the **xxx_ScrInit**:

```
#include <gai/gai.h>
#include <gai/misc.h>
#include <sys/rcm_win.h>
#include <sys/aixgsc.h>
#include <sys/rcmioctl.h>
#include <aixPrivates.h>
extern gAdapterPtr gaiDevAdpIndex[MAXSCREENS];
gsc_handle gscHandle;
Bool xxxScrInit(int index, ScreenPtr pScreen, int argc, char **argv)
{
   gAdapterPtr pAdapter = NULL;
   make_gp makegp_info;
   struct xxx_map xxx_makegp_info;    /* see RCM  */
                                      /* documentation */
   int count = 0;
   char *cmdlist[1] = {(char *) NULL};
   strcpy (gscHandle.devname, aixScreenName(pScreen));
   if (ioctl (aixScreenFD(pScreen), GSC_HANDLE, &gscHandle) < 0) {
   return (FALSE);
   }
   pAdapter = CreateAdapter (aixDeviceID(pScreen),
               gscHandle.handle, count, cmdlist);
   makegp_info.error = 0;
   makegp_info.pData = (char *) &xxx_makegp_info;
   makegp_info.length = sizeof(struct xxx_map);
   makegp_info.access = EXCLUSIVE_ACCESS;
                                      /* not a direct access device
*/
   if (aixgsc(gscHandle.handle, MAKE_GP, &makegp_info)) {
              return (FALSE);
   }
   vram_bits = makegp_info.segment +
               ((struct xxx_map *) makegp_info.pData)->vr_addr;
   /* continue with remainder of adapter initialization ,
        return TRUE on success, FALSE otherwise.  */
}
```

Minimum Resource Management Subsystem (RMS) for 2D Adapters

The Resource Management Subsystem (RMS) is responsible for managing resources common to different rendering models. These resources include cursors, color palettes, buffers, client clip information, fonts, window geometries, and so on. It is not necessary to provide full-function RMS support for an adapter that supports only the X rendering model, as X handles its own resources.

In order to provide the minimal RMS support required by the Graphics Adapter Interface (GAI) architecture, the 2D graphic adapter load module must call the RMS **CreateAdapter** function in the adapter initialization subroutine and the RMS **DestroyAdapter** function in the adapter uninitialize or destroy subroutine.

Implementation

Adapters that are not direct access adapters and will never be direct access adapters can write an abbreviated version of the Resource Management Support. The following calls must be made by a 2D adapter that is intended to display any of the SOFT 3D APIs, such as OpenGL or SoftPHIGS. Otherwise, these calls are recommended, but not required.

CreateAdapter This routine is the basic initialization routine of the GAI RMS library. It provides the necessary initialization to create a graphics process within the RCM (Rendering Context Manager) and permits the graphics process to begin configuration and operation. The routine is typically called in the ddx specific adapter initialization routine.

DestroyAdapter This routine destroys all data structures allocated for the adapter and conveys information to the RCM that invalidates the graphics process' adapter access privileges. This routine is typically called in the ddx specific routine vectored through when the X server terminates or resets.

Adapters that use the minimal RMS will not physically have an RMS load module and will store the value **NORMS** in the rms ODM entry.

The calling sequence for the **CreateAdapter** and **DestroyAdapter** functions are as follows:

```
gAdapterPtr CreateAdapter(int aid, GSC_HANDLE gsc_handle, int
argc, char** argv);
```

The **CreateAdapter** function is called from the **xxxScrInit** subroutine.

```
pAdapter->pProc->DestroyAdapter(pAdapter);    /* pAdapter is */
      /* the adapter pointer returned from CreateAdapter */
```

The **DestroyAdapter** function is typically called from the **xxxCloseScreen** routine.

Include Files

In addition to the standard X server include files, the compilation units (c source files) which contain the implementation of the device dependent initialization and uninitialization routines must include the following header files:

- **<sys/rcm_win.h>**
- **<gai/gai.h>**
- **<gai/misc.h>**
- **<sys/aixgsc.h>**
- **<sys/rcmioctl.h>**
- **<X11/ext/aixPrivates.h>**

These header files resolve type definitions for GAI specific data types and external function declarations for GAI-specific functions used by 2D graphics adapter load modules and input adapter load modules. In addition, they make available the use of the AIX Graphics System Call (aixgsc). At initialization time, an aixgsc call must be made to the device independent

portion of the RCM to register the process (in the case of a 2D graphics adapter ddx module bound with the X server, the X process) as a graphics process.

Configuring the 2D Adapter into the ODM Database

Information about where loadable modules exist is stored in the ODM database. These records are located in the GAI class of the database in /**usr**/**lib**/**objrepos**. At load time, this database is queried based on a unique adapter ID for the correct modules to load. There are two entries that are required for 2D graphics adapters: one for the RMS module and one for the DDX module.

Since it is not necessary to provide full-function RMS support for 2D adapters (see previous section), a special keyword "NORMS" has been developed to provide the minimal level of RMS support; "NORMS" should be placed in the Module_Path field of the ODM entry.

The following is the structure definition for the GAI record in the ODM database:

ODM Record

```
class GAI {
 long Adapter_Id;
 char Module_Key[11];
 vchar Module_Path[256];
 vchar Processor_Id[16];
};
```

Field definitions for the GAI record in the ODM database.

Adapter_id Specifies the GAI adapter ID for the adapter. New adapters are assigned this value during development. This is obtained from the display_id attribute of the ODM PdDv Object Class and stored in the *dev_id* field of the appropriate **display_rec** structure when the server initializes.

The ID is given a value of the form 0x04xxmmnn, where:

04	Fixed.
xx	An adapter-specific ID.
0x00 – 0x7f	Reserved for use by the provider of AIX.
0x80 – 0xff	Provided for vendor adapters.

Note: To avoid duplication of adapter IDs within this range, it is advised that you contact the provider of AIX for a list of available IDs and not rely on using values not currently installed in the database.

mm	Specifies the adapter state for vendor adapters. 0x00 indicates that the adapter is functional. No other value is allowed at this time.
nn	Differentiates between multiple instances of the same adapter type (max 4).

Module_Key Specifies the key for the module. Typically, this key is “rms”, “ddx” or one of the 3D or extension keys.

Module_Path Specifies the path name of the load module for this key and adapter. The module path is used along with the **GAI_PATH** environment variable to find the load module.

Processor_Id Identifies the processor. Intended for future use to enable the developer to load modules based on processor type. Currently, the only supported value of this field is 0 (zero), the default, defined as ANY_PROCESSOR.

Usually, ODM records are gathered in a file to be added at configuration time. For example, a samp2d.add file for an adapter with an ID of 0x4330000 may contain the following records:

```
GAI:
Adapter_Id      = 0x4330000
Module_Key      = ”rms”
Module_Path     = ”NORMS”
Processor_Id    = 0

GAI:
Adapter_Id      = 0x4330000
Module_Key      = ”ddx”
Module_Path     = ”sampddx/loadddx”
Processor_Id    = 0
```

Porting Input Devices

This section describes how to add a new input device to the X Server using the X11 Input Extensions. It consists of the following areas:

Input Device Driver Overview

The AIX system supports only keyboards and displays as part of the low-function Terminal (LFT) subsystem. Typically, the X server uses the mouse as its core pointer, but it could also be a tablet or other device that has a valuator with at least two axes and one pointer. Devices with valuators of two or more axes are supported on the system as extension input devices. The AIX system provides built-in load modules for supported extension input devices. Unsupported input devices require their own load modules.

The X server accesses extension input devices through the X11 Input Extension. This information describes adding a new input device to the AIXwindows X server. You must understand how the X11 Input Extension works before trying to port a new input device. This information should be used with the *X11 Input Extension Porting* document from the X Consortium.

The three requirements for adding a new extension input device to the AIXwindows X server are:

- A device driver (or standard tty device driver)
- A load module that contains the device-dependent subroutines for accessing the extension input device
- Entries in the Object Data Manager (ODM) database

Device Driver

The device driver is the interface between the X server and the extension input device hardware. If the extension input device can use the serial port, you may want to use the standard tty device driver rather than write a device driver. The requirements for the device driver vary, depending upon whether you want to use the X server input ring. If you use the standard tty device driver, you may need to maintain your own input ring. For information on using the tty device driver, see “STREAMS-Based TTY Subsystem Interface” on page 11-1.

To use the X Server input ring to store events, follow the specification for adding events to the input ring. The device driver must send a signal to the X server to notify it that there is input pending. The load module for the extension input device is responsible for sending the input ring information of the X server to the device driver. The **deviceProc** subroutine must initialize the **inputInfoDevPriv** structure for the extension input device.

If you do not want to use the X server input ring, then the load module and the device driver must coordinate how and where input device events are to be stored. The load module must register with the X server the two values it will use to check for input pending. The module must also determine the subroutine to call if input is available. This can be done using **AddInputCheck** subroutine. You will need to follow this method if you plan on using the standard tty device driver. You must also register a block handler and a wakeup handler.

InputInfo Structure

The **InputInfo** structure, defined in the /**usr**/**include**/**X11**/**ext**/**inputstr.h** file, contains information about the input devices. The **inputInfo** global variable must be used to access the elements of the structure.

The **InputInfo** structure is defined as follows:

```
typedef struct {
  int             numDevices;  /* total number of devices      */
  DeviceIntPtr    devices;     /* all devices turned on        */
  DeviceIntPtr    off_devices; /* all devices turned off       */
  DeviceIntPtr    keyboard;    /* fast path to keyboard device */
  DeviceintPtr    pointer      /* fast pointer to the pointer  */
                               /* device                       */
#ifdef AIXV4
  inputInfoDevPriv infoPriv;   /* the global input specific    */
                               /* control block                */
#endif
} InputInfo;
```

The fields of the **InputInfo** structure are defined as follows:

`numDevices` Specifies the number of devices that are known to the X server.

`devices` Specifies a pointer to a linked list of device structures that represent devices that are turned on.

`off_devices` Specifies a pointer to a linked list of device structures that represent devices that are turned off.

`keyboard` Specifies a pointer that is a fast path to the core keyboard structure.

`pointer` Specifies a pointer that is a fast path to the core pointer structure.

`infoPriv` Specifies a pointer to the private structure for the input control blocks.

inputInfoDevPriv Structure

The **InputInfo** structure contains a private area that is used for input control blocks. The following structure is used as the **InputInfo** private area. The **MAX_DEVICES** variable is defined in the **inputstr.h** file. Use the **infoInput** global variable to access this structure. The **devices** and **eventHandlers** elements must be set in the **deviceProc** subroutine if the X server input ring is to be used to store the device events.

```
struct _inputInfoDevPrev {
      struct  inputring      *inputRing;
      struct  DeviceIntRec   *devices [MAX_DEVICES] ;
      void                   (*eventHandlers [MAX_DEVICES]) () ;
} inputInfoDevPriv,  *inputInfoDevPrivPtr;

#define INPUT_RING_SIZE 4096
```

The fields of the **inputInfoDevPriv** structure include are defined as follows:

`inputRing` Specifies a pointer to the input ring where the input devices store events. The **inputring** structure is defined in system include file /**usr**/**include**/**sys**/**inputdd.h**.

`devices` Specifies an array of pointers to the **DeviceIntRec** structure for each device and serves as a fast path to the structures for each device. The index into the array is the device ID stored in the **DeviceIntRec** structure for the extension input device.

`eventHandlers` Specifies an array of subroutines (one for each possible device) that format the raw event as read from the input ring into the proper X server format. The index into the array is the device ID stored in the **DeviceIntRec** structure for the extension input device.

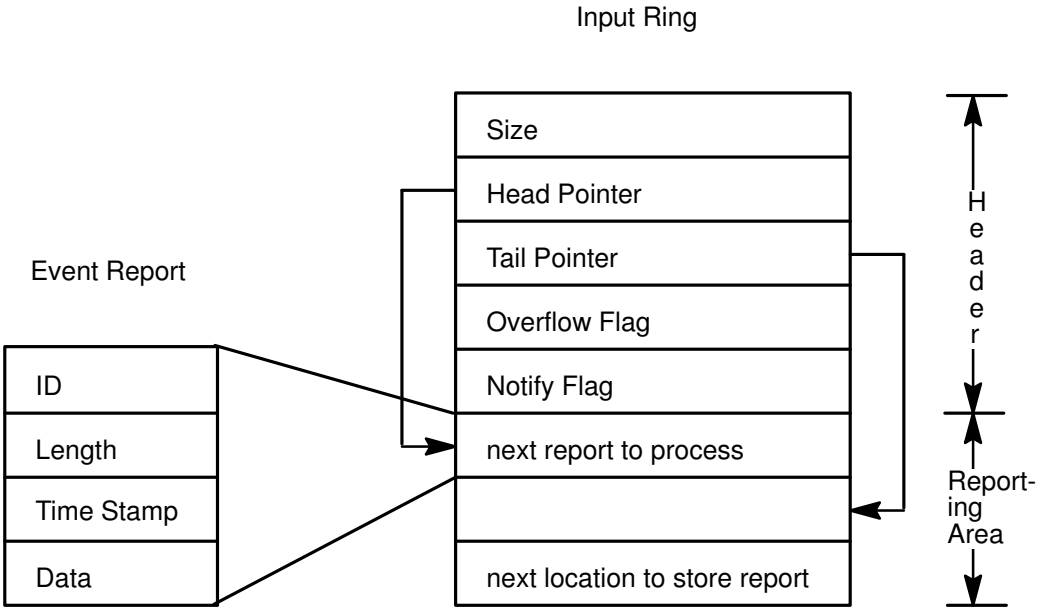
inputring and ir_report Structures

The **inputring** and **ir_report** structures are defined in the **inputdd.h** include file. Refer to the **inputdd.h** file for details of the structure. The first 10 bytes in each **ir_report** structure contain the ID of the device returning the event, the size of the event, and a time stamp. The Input Ring Structure figure depicts the location of this data within the context of the input ring.

X Server Input Ring

Once a device has been opened, extension input device drivers can add events to the X server input queue. The input ring is allocated in X server memory and is stored as part of the device private area of the input control block in the **inputInfo** structure. To use the input queue, the **deviceProc** subroutine for each extension input device must register the input ring data with the device driver. This data is the pointer to the **inputring** structure and the device ID. The device ID for each device is the `id` field of its **DeviceIntRec**.

See the following diagram that illustrates the X Server input ring.



Input Ring Structure

All input devices that receive input events from device drivers by this mechanism share the input ring. The input queue contains a head and tail pointer. The device driver adds device events to the ring at the tail pointer, and the X server reads events from the ring at the head pointer. The header of each raw device event contains a device ID, the size of the event data, and a timestamp in milliseconds. The offset to the next event is computed from size value. The locking strategy uses priority levels to ensure that only one input driver touches the input ring at any given time. Thus, all input device drivers must execute with an **INTCLASS3** priority whenever they modify the input ring.

When the queue head pointer is equal to the tail pointer, it is assumed that the queue is empty. When the device driver attempts to add an event that would cause the tail pointer to be equal to or logically greater than the head pointer, it sets the **ir_overflow** flag and flushes the input event that caused the overflow. Additionally, all events are flushed as long as the **ir_overflow** flag in the ring header is set to **IROVERFLOW**, even if there is room on the ring for an event report.

Data stored in the input ring are in the form of raw device events, but they must be reformatted into the type of event that the X server returns to the client. The events may be core device events (those originating from a pointer or keyboard) or extension device events, such as valuator motion, button press, button release, key press, key release or

proximity. Since the X server may not have knowledge of every raw device event format stored in the queue, it calls a device-specific subroutine to reformat the event into one of the event types that the X server understands. These subroutines are stored in the input device private structure in the **processRawInputEvents** field.

A fast path to each of the extension input device's **processRawInputEvents** is the **eventHandlers** array stored in the private area of the **inputInfo** structure. When the device is opened, the **deviceProc** subroutine stores a pointer to the device's **processRawInputEvents** subroutine in the **eventHandlers** index that corresponds to the device's ID as stored in the `id` field of the **deviceIntRec** structure. The **processRawDeviceEvents** formats the event into the appropriate X events and calls the subroutine stored in the **processInputEvents** field of the **DeviceIntRec** for the extension input device.

SIGMSG Signal

Input device drivers using the AIX input event queue provide event notification to the X server using the **SIGMSG** signal. The **SIGMSG** signal is defined in the **/usr/include/sys/signal.h** file. The **ir_notifyreq** flag in the input ring header specifies when the **SIGMSG** signal is to be sent. If the flag is set to **IRSIGEMPTY**, the driver sends a signal only when it enqueues an event report into an empty input ring. If the **ir_notifyreq** flag is set to **IRSIGALWAYS**, then the device driver sends a **SIGMSG** signal each time it enqueues an event.

Notification is provided for the event that causes a queue overflow; however, once queue overflow has been posted to the input ring, no other event notification is provided until the overflow indicator has been cleared. Queue overflow is posted to the input ring when the **ir_overflow** flag is set to **IROVERFLOW**.

If the input device to be added connects to the system through a tty port, a block and wakeup handling scheme must be used. See "Block and Wakeup Handling" on page 12-50 for more information.

Block and Wakeup Handling

If you use the standard tty device driver, you must notify the X server to enable the extension input device driver for input pending. This can be accomplished by calling the **AddEnabledDevice** subroutine, which adds the file descriptor for the device driver to the select mask. To remove the file descriptor for the device driver from the select mask, use the **RemoveEnabledDevice** subroutine. Both subroutines should be called in the **deviceProc** subroutine for the extension input device when the extension input device is turned on or off. The **xxxBlockHandler** and **xxxWakeupHandler** subroutines must be registered with the X Server by calling the **RegisterBlockAndWakeupHandlers** subroutine to control what the X Server should do before and after calling the **select** subroutine.

Note: You must replace the **xxx** portion of the subroutine name to make it unique.

The **xxxBlockHandler** and **xxxWakeupHandler** subroutines are removed by calling **RemoveBlockAndWakeupHandlers** when the extension input device is turned off in a **deviceProc** subroutine of the device.

The **AddEnabledDevice**, **RemoveEnabledDevice** and **RemoveBlockAndWakeupHandlers** subroutines are common X Server routines. For more information about these subroutines, refer to *The X-Window Server* by Elias Israel and Erik Fortune, Digital Press.

Additional information about the **xxxBlockHandler** and **xxxWakeupHandler** subroutines follows.

xxxBlockHandler Subroutine

Purpose

Allows for any device-specific action before the **select** subroutine is processed.

Syntax

void xxxBlockHandler (*blockData***, *ppTimeout*, *pReadmask***)**
char ***blockData*;**
struct timeval ****ppTimeout*;*
unsigned ***pReadmask*;**

Description

The **xxxBlockHandler** subroutine allows for any device-specific action before the X server processes the **select** subroutine. The **xxxBlockHandler** subroutine is called before the X server calls the **select** subroutine. For most extension devices, this subroutine is an empty routine. You must register block and wakeup handlers in pairs, and in most cases, only a wakeup handler is needed for input devices.

Parameters

blockData	Points to the private data used by the block handler subroutine. Its value is the same as the value of the **blockData** parameter that was passed to the **RegisterBlockAndWakeupHandlers** subroutine.
ppTimeout	Returns a pointer to the timeout value. This value is used to determine the maximum amount of time the block is allowed to last.
pReadmask	Points to a data structure that defines which descriptors are to be blocked on. The data structure is an array of unsigned long data elements, and the bit in the arrary must be set so that it corresponds to the value of the file descriptor.

xxxWakeupHandler Subroutine

Purpose

Allows for reading from ready file descriptors after **select** processing.

Syntax

#define FD_SETSIZE 128 /* limit of server connections */
#include <sys/select.h>
void xxxWakeupHandler (*result***, *pReadmask***)**
int *result***;**
unsigned ***pReadmask*;**

Description

The **xxxWakeupHandler** subroutine is called after the X server returns from the **select** subroutine. If the descriptor for the extension input device driver is ready for reading, the two long values that check for input should be set not equal to each other.

The macro **FD_ISSET(***n***, *pReadMask***)** checks the file descriptor that you are using to see if the **select** subroutine selected your ID.

Parameters

result	Indicates the return value from the **select** subroutine that specifies the number of descriptors that are ready for reading.
pReadMask	Points to a data structure that defines which descriptors are ready for reading.

Event Processing

Processing events involves obtaining events from devices and delivering them to clients. If the X server input ring is used, each device event in the input ring is passed to the **eventHandlers** subroutine, which was initialized in the **inputInfoDevPriv** structure. Otherwise, the input processing subroutine passed to the **AddInputCheck** subroutine will be called to process pending input. Both of these methods should use the **processRawEvents** subroutine, which is part of the load module.

AddInputCheck Subroutine

Purpose

Adds additional input checking for extension input devices that are not using the X server input ring.

Syntax

void AddInputCheck (*p1***,** *p2***,** *proc***)**
void ****p1*, ****p2***;**
void ****proc* **();

Description

The **AddInputCheck** subroutine adds extra input checking for extension input devices that do not use the X server input ring. The **AddInputCheck** subroutine adds a level of indirection to the current scheme of checking for input by comparing of two long data elements. Instead of one set of long data elements to check for pending input, there is an array of longs. The **AddInputCheck** subroutine adds a new set of long data elements as well as an input processing procedure into this array.Two identical values indicates no input whereas two different values indicates input is pending. The long values can be pointers or hard-coded values.

Parameters

p1, *p2*	Specifies a pointer to a long data element. The values can be a mask, a hard coded value, an integer, or a pointer.
proc	Specifies a procedure to call if the values pointed to by the *p1* and *p2* parameters are not equal.

RemoveInputCheck Subroutine

Cancels the **AddInputCheck** subroutine.

Syntax

void RemoveInputCheck (*proc***)**
void ****proc* **();

Description

The **RemoveInputCheck** subroutine removes additional input checking for extension input devices that do not use the X server input ring. This subroutine is called when the **deviceProc** subroutine of the device contains the **DEVICE_OFF** value.

Parameters

proc	Specifies the procedure that corresponds to the input check that should be removed.

Input Load Module

The subroutines that manipulate each extension input device are stored in a dynamically loadable module. When the X server receives an **X_ListInputDevices** protocol request, it loads each of the extension load modules and calls the entry point for each load module. Load modules remain loaded until the X server terminates or resets.

InputDevPrivate structure

The **InputDevPrivate** structure must be initialized in the entry point subroutine of the extension input device load module. This structure is part of the **/usr/include/X11/ext/aixInput.h** file and is defined as follows:

```
typedef struct {
    pointer  client_list;
    short    ctrlstructsize;
    int      (*legalModifier) ();
    int      (*deviceProc) ();
    int      (*openDevice) ();
    void     (*closeDevice) ();
    int      (*setDeviceMode) ();
    int      (*setDeviceValuators) ();
    int      (*getDeviceControl) ();
    int      (*changeDeviceControl) ();
    void     (*processRawInputEvents) ();
    int      fd;
    int      idx;
    pointer  vendor_specific;
} InputDevPrivate;
```

Field definitions for the **InputDevPrivate** structure:

`client_list` Specifies a linked list of all the clients that have this device open. This element must be initialized to NULL.

`ctrlstructsize`
Specifies the size of the device control structure that this device accepts. This field is used in the **ChangeDeviceControl** protocol. If the extension input device does not support changing controls, this element must be set to 0.

`legalModifier`
Specifies a pointer to a subroutine that is used for devices that have keys. This field returns **True** or **False** if the key passed as an input parameter can be used as a modifier key. If the extension input device does not use a modifier key, this element must be set to **False**.

`deviceProc` Specifies a pointer to the **deviceProc** subroutine. The **deviceProc** subroutine initializes the device, turns the device on and off, and closes it.

`openDevice` Specifies a pointer to the subroutine to call when the **OpenDevice** protocol is specified.The initial value of this field is **NoopDDA()**.

`closeDevice` Specifies a pointer to the subroutine to call when the **CloseDevice** protocol is specified.The initial value of this field is **NoopDDA()**.

`setDeviceMode`
Specifies a pointer to the subroutine to call when the **SetDeviceMode** protocol is specified. If the extension input device does not support mode changing, this element must be set to NULL.

`setDeviceValuators`
: Specifies a pointer to a subroutine to call when the **SetDeviceValuators** protocol is specified. If the extension input device does not support valuators or initializing of valuators, this element must be set to NULL.

`getDeviceControl`
: Specifies a pointer to a subroutine to obtain the current device control setting for the device. If the extension input device does not support querying of device controls, this element must be set to NULL.

`changeDeviceControl`
: Specifies a pointer to a subroutine to set the device control settings for a device. If the extension input device does not support changing device controls, this element must be set to NULL.

`processRawInputEvents`
: Specifies an input subroutine that formats the raw events extracted from the input ring.

`fd`
: Specifies the file descriptor of the extension input device.

`idx`
: Specifies an index to device-specific information. Do not initialize this field.

`vendor_specific`
: Specifies a pointer to any vendor private data used in the load module of the extension input device.

ExtInitInput Subroutine

Purpose

Initializes the input private structure for the extension input device.

Syntax

int ExtInitInput (*inputDevPrivate***)**
InputDevPrivate ****inputDevPrivate*;**

Parameters

inputDevPrivate Specifies a pointer to an empty input device private structure and returns an initialized structure.

Description

The **ExtInitInput** subroutine for the extension input device initializes the **devPrivate** structure of the **deviceIntRec** structure. The **ExtInitInput** subroutine is the entry to the load module and is returned from the **load** system call.

Return Values

0 Indicates successful completion.

1 Indicates an error occurred.

deviceProc Subroutine

Purpose

Initializes, turns on, opens, or closes the extension input device.

Syntax

int xxxdeviceProc (*pDev***,** *onoff***)**
DevicePtr *pDev***;**
int *onoff***;**

Description

The **deviceProc** subroutine is a device-specific procedure that is called from the **EnableDevice**, **DisableDevice**, **InitAndStartDevices**, **CloseDevice**, and **ExtInitInput** subroutines. This subroutine performs the function specified by the *onoff* parameter for the device specified by the *pDev* parameter.

Typical actions that occur during device initialization include setting up device private information and calling the **dix** subroutines to initialize device-specific information. These subroutines include **InitializePointerDeviceStruct** and **InitializeKeyboardDeviceStruct** to initialize the pointer and keyboard. The **deviceProc** subroutine must initialize extensions, device class structures, and the global state of the device. The extension class device structures are created by using the following subroutines:

- **InitButtonClassDeviceStruct**
- **InitKeyClassDeviceStruct**
- **InitValuatorClassDeviceStruct**
- **InitValuatorAxisClassDeviceStruct**
- **InitFocusClassDeviceStruct**
- **InitProximityClassDeviceStruct**
- **InitKbdFeedbackClassDeviceStruct**
- **InitPtrFeedbackClassDeviceStruct**

The **MakeAtom** subroutine is called with the name of the device. The **AssignTypeAndName** subroutine is called with the device ID and the atom returned from the **MakeAtom** subroutine. These subroutines set the device name and type in the device structure.

The **deviceProc** subroutine must initialize the default values for global state in the **devPrivate** section of the **DeviceIntRec** structure for the specified device.

Parameters

pDev Specifies a pointer to the structure that devices the device

onoff Specifies the action to be taken. Valid values for *onoff* are:

DEVICE_INIT Creates structures.

DEVICE_ON Opens the hardware device driver if one exists. The device is marked as on and calls the **AddEnabledDevice** subroutine. Other OS-specific actions occur to make the device available. Typically, the **deviceProc** subroutine calls the **AddEnabledDevice** subroutine to add the device to the list of “on” devices and the `on` field of the device structure is set to **True**.

If an extension device does not receive its events from input ring, then the **AddInputCheck** and **RegisterBlockAndWakeupHandlers** subroutines are called to set up the input checking and processing mechanism for the extension device.

DEVICE_OFF Marks the device off. Typically, **deviceProc** calls the **RemoveEnabledDevice** subroutine to delete the device from the list of “on” devices and adds the device to the list of initialized devices. The `on` field of the device structure is set to **False**.

If the **RegisterBlockAndWakeupHandlers** subroutine was called during the **DEVICE_ON** state, the **RemoveBlockAndWakeupHandler** subroutine turns off the block and wakeup handlers for the device.

DEVICE_CLOSE
If the **AddInputCheck** subroutine was called during the **DEVICE_ON** stage, the **RemoveInputCheck** subroutine turns off input checking and processing for the extension input device in the server.

Return Values

0	Indicates successful completion.
1	Indicates an error occurred.

setDeviceMode Subroutine

Purpose

Sets the mode of a device.

Syntax

int xxxsetDeviceMode (*pDev***,** *client***,** *mode***)**
DeviceIntPtr *pDev***;**
ClientPtr *client***;**
int *mode***;**

Description

The **setDeviceMode** subroutine sets the mode of the device to the mode passed as an input parameter. This mode can be **Absolute** or **Relative.**

Parameters

pDev	Specifies the device.
client	Specifies the client opening the input device.
mode	Specifies the mode. Valid values are **Absolute** or **Relative**.

Return Values

BadDevice	Indicates an invalid device ID.
BadMatch	Indicates the device does not support modes.
BadMode	Indicates the specified mode is invalid.
DeviceBusy	Indicates another client is using this device.

setDeviceValuators Subroutine

Purpose

Initializes the valuators on a device.

Syntax

int xxxsetDeviceValuators (*pDev***,** *client***,** *valuators***,** *first_valuator***,** *num_valuators***)**
DeviceIntPtr *pDev***;**
ClientPtr *pClient***;**
int ****valuators*;**
int *first_valuator***;**
int *num_valuators***;**

Description

The **setDeviceValuators** subroutine initializes one or more of the valuators of the device to the values passed in the *valuators* parameter.

Parameters

pClient	Specifies the client setting the valuators the input device.
pDev	Specifies the device.
valuators	Specifies an array of valuator values to be set.
first_valuator	Specifies the index of the first valuator to be set.
num_valuators	Specifies the number of valuators to be set.

Return Values

Success

BadMatch	Indicates the device does not permit valuators to be set.
BadValue	Indicates an invalid value for the valuator was specified.
BadDevice	Indicates an invalid device ID was specified.

getDeviceControl Subroutine

Purpose

Returns the current device control setting for the specified device.

Syntax

int xxxgetDeviceControl (*pClient***,** *pDev***,** *control***)**
ClientPtr *client*
DevicePtr *pDev***;**
xDeviceCtl ***control***;**

Description

The **getDeviceControlProc** subroutine for each device obtains the specified controls for the device. The **xDeviceControl** structure is passed directly to the client. The client must link third-party vendor code and initialize the appropriate subroutines to interpret the data**.**

Parameters

pClient	Specifies the client that is requesting the control change.
pDev	Specifies a pointer to the device.
control	Specifies the control to obtain.

Return Values

Success

BadValue	Indicates the X server does not support specified control.
BadMatch	Indicates the X server supports specified control, but the device does not.
AlreadyGrabbed	Indicates the device was grabbed by another client.

changeDeviceControl Subroutine

Purpose

Changes the device control settings for the specified device.

Syntax

int xxxchangeDeviceControl (*pDev***,** *control***)**
DevicePtr *pDev***;**
xDeviceCtl ***control;**

Description

The control procedure for each device changes the specified controls.

Parameters

pDev	Specifies a pointer to the device.
control	Specifies the change to the device control.

Return Values

Success

Failure

processRawInputEvents Subroutine

Purpose

Formats a raw device event into an event that can be sent to an X Client.

Syntax

void xxxprocessRawInputEvents (*pDev***,** *rawEvents***)**
DeviceIntRecPtr *pDev***;**
pointer ***rawEvent;**

Description

The **processRawInputEvents** subroutine takes the raw device event as retrieved from the input queue or input device and formats it into an event that can be sent to an X client.

In order to map raw device event into X event, information about the current screen and cursor position is needed. This type of information is accessible globally as follows:

```
extern int DeviceButtonPress, DeviceButtonRelease;

/* extension device event types */
extern int DeviceMotionNotify, DeviceValuator;

/* extension device event types */
extern int ProximityIn, ProximityOut;

/* extension device event types */
extern int lastEventTime;
extern ScreenInfo screenInfo;   /* screen information */
extern int aixCurrentScreen;    /* current screen number */
extern InputInfo inputInfo;     /* global InputInfo,
                                   defined in dix/devices.c */
extern int AIXCurrentX;         /* current X position of cursor */
extern int AIXCurrentY;         /* current Y position of cursor */
extern long GetTimeInMillis();  /* function to get current time */
```

Parameters

pDev	Specifies a pointer to the **DeviceIntRec** structure for the device.
rawEvent	Specifies a pointer to the raw device event for the device.

ODM Database Entry for Input Devices

Information about each input device is stored in the ODM (Object Data Manager) database. The following information will be stored for each device.

```
XINPUT {
  char  DeviceName[64];          /* VENDOR'S NAME FOR DEVICE   */
  vchar ModuleName[MAXPATHLEN]   /* PATH NAME TO LOAD MOCULE   */
  char  GenericName[15];         /* ONE OF 18 GENERIC DEVICES  */
  vchar Class[16];               /* PdDv CLASS                 */
  vchar SubClass;
  vchar connwhere;
};
```

The fields of the **XINPUT** ODM record structure are defined as follows:

`DeviceName` Specifies a vendor specific name for the device.

`ModuleName` Specifies the relative name of the load module. This name is used in conjunction with the **XINPUTPATH** environment variable to find the name of the **XINPUT** load module. The default path is set to: **/usr/lpp/X11/lib/load**.

`GenericName` Specifies the name of the device as used in name field of the **deviceIntRec** structure. There are 18 predefined names defined in the **XI.h** include file. Vendors are encouraged to use these names for interoperability.

`Class` Same as the class name in the device driver CuDv record.

`SubClass` Reserved for future use.

`connwhere` Reserved for future use.

ODM Input Device Record Example

The **.add** file for the system-supported input devices will contain the information for the input device. For instance, the keyboard device will have the following record:

```
XINPUT:
DeviceName = "Keyboard Device"
GenericName= "KEYBOARD"
ModuleName = "loadkeyboard"
Class      = "Keyboard"
```

Sample Input Device Load Module

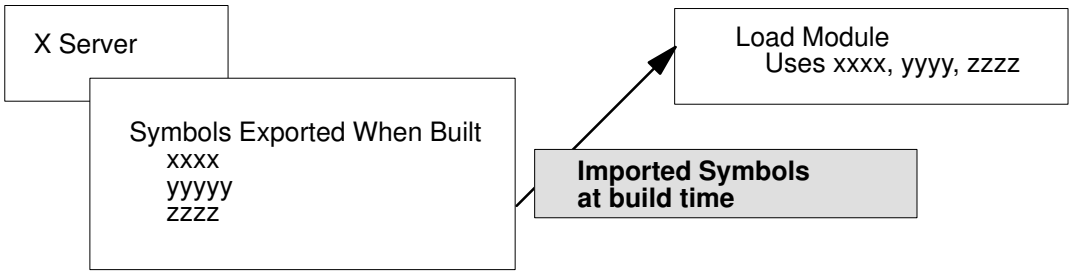
A sample Input Device load module is included in the AIX Version 4.1 distribution and is located in the directory **/usr/lpp/X11/Xamples/extensions/server/load**.

If this directory is not on your system, you may have not loaded the **X11.samples.ext** LPP. To see if this is on your system execute:

```
lslpp -h | grep samples.ext
```

Building a Dynamically Loadable Module

The figure shows how dynamically loadable modules are built.



Build of Dynamically Loadable Modules

To build a dynamically loadable module, the loading module and the loaded module must cooperate. The loading module must export a list of symbols that will be needed by the loadable module. The loadable module must import the symbols needed from the loading module. These symbols are resolved at the time of the **load** system call. In addition, each dynamically loadable module must provide the name of a subroutine to be used as the entry point to this module.

AIX provides the following exports files that can be used by a dynamically loadable module as import files:

X.exp Contains symbols from the device independent (dix) portion of the X server.

Xi.exp Contains symbols from the device independent portion of the X11 Input Extension.

The following example is a sample makefile used to build a dynamically loadable module. This module imports symbols from **X.exp** and **Xi.exp**. The **ExtInitInput** subroutine is used as the entry point to this module:

```
# Makefile for creating loadable object module

# specify entry point
EPNT = SampExtInit

# import files
IMPS =  -bI:/usr/lpp/X11/bin/X.exp\
        -bI:/usr/lpp/X11/bin/Xi.exp

# include file directories
INCS =  -I/urs/include/X11          \
        -I/usr/include/X11/ext      \
        -I/usr/include/X11/extensions

sample:
        cc SampExt.c -o LoadSampExt -e${EPNT} ${INCS} ${IMPS}
-bM:SRE
```

Debugging Load Modules

Remember that symbols defined in the load module are not available to the debugger until the load module is loaded into memory. So, in order to set a breakpoint inside a function defined in the load module, you have to wait until the module is loaded. To do this:

- Get into the debugger (dbx /**usr**/**lpp**/**X11**/**bin**/**X**) and issue a **set** subcommand. This will list all the variables defined in the debug environment.
- Use the **unset** subcommand to unset the variable **$ignoreload**:

  ```
  unset $ignoreload
  ```

- Issue the **run** subcommand at the debug prompt:

  ```
  run -Pll dev_name
  ```

  where *dev_name* is the index of the device you want to debug from the **lsdisp** command (for example, `ppr0`). The process of loading and unloading generates signals which the debugger catches and pauses. At this time you set the breakpoint.

As an example, say that there are two displays on the system:

```
>lsdisp
DEV     SLOT    BUS     APT_NAME        DESCRIPTION
ppr0    00-01   mca     POWER_G4        Midrange Graphics Adapter
gda0    00-03   mca     colordga        Color Graphics Adapter
```

Get into the debugger:

```
dbx /usr/lpp/X11/bin/X                          (user action)
dbx>unset $ignoreload                           (user action)
dbx>run -P11 gda0                               (user action)
```

At this point the LFT subsystem takes over the cursor and you will see a screen flash on the vendor display. Type Shift-Action to get the cursor back to the ibmdisp display.

```
dbx> stopped due to load/unload         (system response)
dbx> stop in xyz                        (user action, xyz() is
                                        the function you want to
                                        break in.)
```

List of X Server Porting Subroutines

Load modules for the AIXwindows X server must contain the following subroutines.

X Server Initialization

The following X Consortium subroutines must be included to meet X server requirements. One instance is required for each server.

ddxProcessArgument
Processes the command-line arguments for the X server.

FindAllAvailableDisplays
Initializes the **DisplayRec** structure for the X server.

InitOutput Initializes the **screeninfo** *ImageByteOrder*, bitmap information, and pixmap format information and obtains private indexes for the screen and AIX extensions.

Device-Dependent Initialization

The following subroutines must be supplied to meet ddx requirements:

xxxentryFunc Returns information from the loading of the loadddx module.

xxxScrInit Initializes the screen structure for each screen and performs device-dependent initialization as required.

xxxCloseScreen
Closes the screen during X server reset.

Block and Wakeup Handling (Input Devices)

Third-party vendors need to write these subroutines for their input extension load modules.

xxxBlockHandler
Allows for any device-specific action before the select subroutine is processed.

xxxWakeupHandler
Allows for reading from ready file descriptors after **select** processing.

Event Processing (Input Devices)

The following are provided callable routines for input extension device event processing:

AddInputCheck
Adds additional input checking for extension input devices that are not using the X server input ring.

RemoveInputCheck
Cancels the **AddInputCheck** subroutine.

Input Load Module (Input Devices)

The following subroutines must be supplied for input extension load modules:

ExtInitInput Initializes the input private structure for the extension input device.

deviceProc Initializes, turns on, opens, or closes the extension input device.

setDeviceMode
Sets the mode of a device.

setDeviceValuators
Initializes the valuators on a device.

getDeviceControl
Returns the current device control setting for the specified device.

changeDeviceControl
Changes the device control settings for the specified device.

processRawInputEvents
Formats a raw device event into an event that can be sent to an X client.

Related Information

Angebranndt, Susan, Drewry, Raymond, Karlton, Philip, Newman, Todd, Packard, Keith and Scheifler, Robert W. *Strategies for Porting the X v11 Sample Server.* Massachusetts Institute of Technology. 1991.

Fortune, Erik, and Israel, Elias. *The X-Window Server*. Digital Press.

Gettys, James, Newman, Ron and Scheifler, Robert W. *Xlib—C Language X Interface, MIT X Consortium Standard, X Version 11, Release 5.* MIT X Consortium 1991.

Patrick, Mark, and Sachs, George. *X11 Input Extension Library Specification. MIT X Consortium Standard. X Version 11, Release 5*. Hewlett-Packard Company, Ardent Computer, and the Massachusetts Institute of Technology. 1989, 1990, 1991.

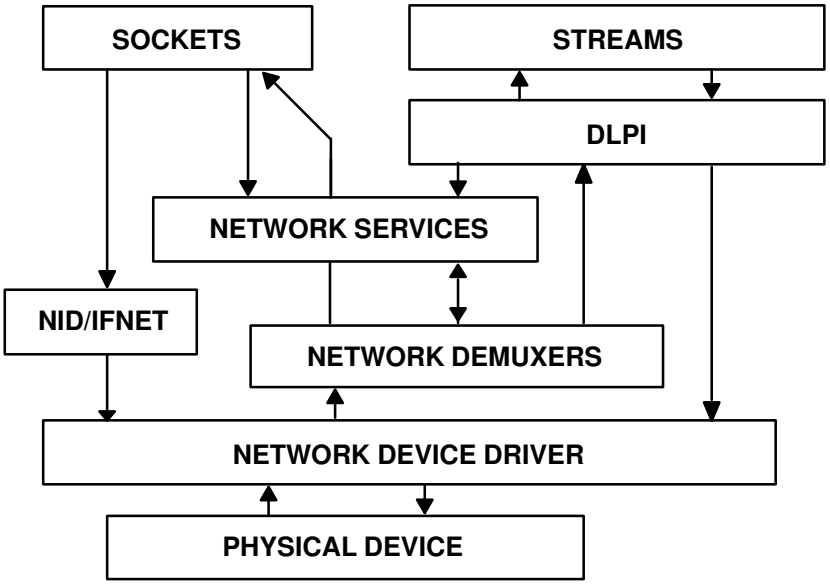
Patrick, Mark, and Sachs, George. *X11 Input Extension Protocol Specification. MIT X Consortium Standard. X Version 11, Release 5*. Hewlett-Packard Company, Ardent Computer, and the Massachusetts Institute of Technology. 1989, 1990, 1991.

Sachs, George. *X11 Input Extension Porting Document. MIT X Consortium Standard. X Version 11, Release 5*. Hewlett-Packard Company and the Massachusetts Institute of Technology. 1989, 1990, 1991.

Scheifler, Robert W. *X Window System Protocol, MIT X Consortium Standard, X Version 11, Release 5.* MIT X Consortium 1991.

Chapter 13. Implementing a Network Device Driver

A Common Data Link Interface (CDLI) device driver has several components, as shown in the CDLI Device Driver Structure figure.



CDLI Device Driver Structure

Several of those components are system-supplied, but a device driver writer typically writes the following components:

Network Device Driver (NDD)
: Defines a simple interface to network based devices that can be used by both the sockets network interface layer (IFNET) and the STREAMS DLPI data link layers. (See “Writing a Network Device Driver” on page 13-2.)

Network Demuxer (ND)
: Provides common data-link receive functionality. The demuxer specifies receive filters that are used to distribute network packets. (See “Writing a Network Demuxer” on page 13-12.)

Network Interface Driver(NID)
: The AIX Network Interface Driver (NID) is a layer of software between a network device driver (NDD) and an AIX network layer. This layer is required for all network device drivers that have to be made available to a network layer. (See “Writing a Network Interface Driver” on page 13-21.)

Writing a Network Device Driver

Network device drivers, including the system provided network device drivers (for example, Ethernet and Token-Ring), are implemented as loadable kernel extensions in AIX. The following general guidelines apply:

- By convention, device driver kernel extensions are installed in the **/usr/lib/drivers** directory. Some system utilities may assume that this is the case.
- When building the kernel extension the relevant base system exports must be imported. The system provided device drivers import the following: **kernex.exp**, **syscalls.exp**. (**streams.exp** is used by some drivers). These system export files are located in the **/usr/lib** directory.
- Device driver writers must decide how much of the kernel extension to pin. The major consideration is that code executed on the interrupt level should be pinned. In general, the system provided device drivers are pinned in their entirety.
- Network device drivers must be configured and loaded into the system.

Overview of Network Device Driver Changes in AIX Version 4.1

To accommodate the changes for the CDLI interface that the communications subsystem now uses, the network device driver's interfaces in AIX Version 4.1 are different from the interfaces in AIX Version 3.2. The parameters passed to the open, close, ioctl and output functions have been changed. The ioctl function has a new list of commands to service. The receive interrupt routine is now required to pass up frames via a required interface. There are also some minor changes to the way a network device driver is loaded and terminated.

Note: An attribute "bnc_dix_jumper", which takes values "yes" or "no", is used for SMIT to prompt a user to change a jumper on an Ethernet card if it is hardware configurable for BNC or DIX cables. This attribute is used in AIX Version 4.1 and in some later versions of AIX Version 3.2.

Network Device Driver Initialization and Termination

Device driver initialization involves execution of the kernel extension's configuration entry point. This function is designated when the network device driver is built and is called by the system when the device driver is loaded. Usually, the **ifconfig** command is responsible for configuring the network device driver. The format of the entry should be as follows:

```
(*networkdd_config) (int cmd, struct uio *uio)
```

The entry has the following parameters:

- `cmd` indicates what type of configuration operation should be performed. The network device driver should recognize the following values for `cmd` (see **/usr/include/sys/device.h**) and can define additional device specific commands:

CFG_INIT	Initialize the device.
CFG_TERM	Terminate the device.
CFG_QVPD	Return vital product data.
CFG_UCODE	Download microcode.

- `uio` is a pointer to a **uio** structure whose data area contains a **ndd_config_t** structure (refer to **/usr/include/sys/ndd.h**). This **ndd_config_t** structure contains the configuration information for the network device driver

When called with the CFG_INIT command the device driver's configuration function should perform the following tasks:

- Do any pinning of modules or data structures required by the device driver.

- Do any lock initialization required by the driver.
- Initialize the device control structures, including allocating memory for the **ndd** structure.
- Call the **ns_attach** kernel service to add the device driver to the list of available network devices.

When called with the CFG_TERM command the device driver's configuration function should perform the following tasks:

- Do any unpinning of modules or data structures required by the device driver.
- Free any lock resources.
- Free any device control structures.
- Call the **ns_detach** kernel service to remove the device driver from the list of available network devices.

Attention: DLPI or the socket network interface layer may still have references to the device driver's **ndd** structure. This structure should only be deleted from the system when the device driver is certain that these references have been removed.

When called with the CFG_QVPD command the device driver's configuration function should return the devices vital product data in the **ndd_config_t** structure.

CFG_UCODE is device specific.

The configuration entry point can be called from the process environment only.

Sample network device driver code for configuration and unconfiguration follows:

```
xx_config(
  int             cmd,              /* command being processed */
  struct uio      *p_uio)           /* pointer to uio structure */
{
  xx_dev_ctl_t *p_dev_ctl = NULL;   /* point to dev_ctl area */
  int rc = 0;                       /* return code */
  int i;                            /* loop index */
  ndd_config_t  ndd_config;         /* config information */


  /*
   * Use lockl operation to serialize the execution of the config commands.
   */
 lockl(&CFG_LOCK, LOCK_SHORT);
 if (!xx_inited) {
         /* perform first time initialization
          *
          *  set all locks
          *  init device driver structures
          */
  }
  pincode(xx_open);                 /* pin the entire driver */

  uiomove((caddr_t) &ndd_config, sizeof(ndd_config_t), UIO_WRITE, p_uio);

  /*
   * find the device in the dev_list if it is there
   */
  p_dev_ctl = xx_dd_ctl.p_dev_list;
  while (p_dev_ctl) {
         if (p_dev_ctl->seq_number == ndd_config.seq_number)
                  break;
         p_dev_ctl = p_dev_ctl->next;
  }
  switch(cmd) {
         case CFG_INIT:
                  if (p_dev_ctl) {
                           rc = EBUSY;
                           break;
```

```
        }
        /*
         * Allocate memory for the dev_ctl structure
         */
        p_dev_ctl = xmalloc(sizeof(xx_dev_ctl_t), MEM_ALIGN,
                 pinned_heap);

        bzero(p_dev_ctl, sizeof(en3com_dev_ctl_t));

        /*
         *  Initialize the locks in the dev_ctl area
         */

        .....
        /*
         * Add the new dev_ctl into the dev_list
         */
        p_dev_ctl->next = en3com_dd_ctl.p_dev_list;
        xx_dd_ctl.p_dev_list = p_dev_ctl;
        xx_dd_ctl.num_devs++;
        /*
         *  Copy in the dds for config manager
         */
        if (copyin(ndd_config.dds, &p_dev_ctl->dds,
                 sizeof(en3com_dds_t))) {
               rc = EIO;
                 break;
        }
        p_dev_ctl->seq_number = ndd_config.seq_number;
        /* save the dev_ctl address in the NDD correlator field */
        p_dev_ctl->ndd.ndd_correlator = (caddr_t)p_dev_ctl;
        p_dev_ctl->ndd.ndd_addrlen = XX_NADR_LENGTH;
        p_dev_ctl->ndd.ndd_hdrlen = XX_HDR_LEN;
        p_dev_ctl->ndd.ndd_physaddr = WRK.net_addr;
        p_dev_ctl->ndd.ndd_mtu = XX_MAX_MTU;
        p_dev_ctl->ndd.ndd_mintu = XX_MIN_MTU;
        p_dev_ctl->ndd.ndd_type = NDD_ISO???;
        p_dev_ctl->ndd.ndd_flags = NDD_BROADCAST | NDD_SIMPLEX;
        p_dev_ctl->ndd.ndd_open = xx_open;
        p_dev_ctl->ndd.ndd_output = xx_output;
        p_dev_ctl->ndd.ndd_ctl = xx_ctl;
        p_dev_ctl->ndd.ndd_close = xx_close;
        p_dev_ctl->ndd.ndd_specstats = (caddr_t)&(XXSTATS);
        p_dev_ctl->ndd.ndd_speclen = sizeof(XXSTATS);

          /* perform device-specific initialization */
          .....
          /* add the device to the NDD chain */
          if (rc = ns_attach(&p_dev_ctl->ndd)) {
                 return(rc);
          }
        break;
case CFG_TERM:
        /* Does the device exist? */
        if (!p_dev_ctl) {
                 rc = ENODEV;
                 break;
        }

        /*
         * Make sure the device is in CLOSED state.
         * Call ns_detach and make sure that it is done
         * without error.
         */
        if (p_dev_ctl->device_state != CLOSED || ns_detach(&(p_dev_ctl->ndd))) {
                 rc = EBUSY;
                 break;
        }
        /*
         * Remove the dev_ctl area from the dev_ctl list
         * and free the resources.
```

```
                 */
                break;
        case CFG_QVPD:
                /* Does the device exist? */
                if (!p_dev_ctl) {
                        rc = ENODEV;
                        break;
                }
               if (copyout((caddr_t)&p_dev_ctl->vpd, ndd_config.p_vpd,
                        (int)ndd_config.l_vpd)) {
                     rc = EIO;
                }
                break;
        default:
                rc = EINVAL;
  }
  /* if we are about to be unloaded, free locks */
  if (!xx_dd_ctl.num_devs) {
        /* free locks here */
        xx_inited = FALSE;
  }
 unpincode(xx_open);         /* unpin the entire driver */
 unlockl(&CFG_LOCK);
 return (rc);
}
```

CDLI – Device Driver Interface

In AIX Version 4.1, network device drivers should be entered only through the kernel CDLI users. STREAMS and sockets are implemented above CDLI, so that users gain access to the network device drivers through the standard socket and STREAMS application interface. The kernel establishes a **ndd** structure (refer to /**usr**/**include**/**sys**/**ndd.h**) for all network devices. This structure defines the entry points that the device driver must support. The following is a list of these entry points:

- **ndd_open**
- **ndd_close**
- **ndd_output**

ndd_open Entry Point

This entry point has the following format:

```
ndd_open(struct ndd *ndd)
```

The parameter `ndd` is a pointer to the system **ndd** structure for this network device. The file /**usr**/**include**/**sys**/**ndd.h** contains the definition of this structure.

Device driver users (DLPI or the socket network interface layer) gain access to the device through a call to the **ns_alloc** kernel service. This kernel service opens the network device, if required, by a call to the **ndd_open** entry point.

When the **ndd_open** function is called, the device driver should allocate the necessary system resources (such as DMA channel, interrupt level and priority). It should register it's interrupt handler with the system using the **i_init** kernel service and initialize the device. After the open is successful, the device driver is responsible for ORing the `ndd_flag` field with NDD_UP. The device driver should have ORed this field with NDD_RUNNING upon successful initialization.

ndd_open can be called from the process environment only.

A code fragment for a sample network device driver open routine follows:

```
xx_open(
  ndd_t         *p_ndd)         /* pointer to the ndd in the dev_ctl area */
  {
   xx_dev_ctl_t   *p_dev_ctl = (xx_dev_ctl_t *)(p_ndd->ndd_correlator);
   int rc;
  /*
   * Set the device state and NDD flags
   */
   p_dev_ctl->device_state = OPEN_PENDING;
   p_ndd->ndd_flags = NDD_BROADCAST | NDD_SIMPLEX;
   /* set up locks, register interrupt handler */
   .....
   .....
   p_ndd->ndd_flags |= (NDD_RUNNING | NDD_UP);
   return(0);
}
```

ndd_close Entry Point

This entry point has the following format:

```
ndd_close(struct ndd *ndd)
```

The parameter `ndd` is a pointer to the system **ndd** structure for this network device.

DLPI or the socket network interface layer relinquishes access to a network device by calling the **ns_free** kernel service. This kernel service will close the device, when the user reference count reaches 0, by a call to the **ndd_close** entry point.

When the **ndd_close** function is called, the device driver should free its system resources (including all mbufs) and deregister its interrupt handler through a call to the **i_clear** kernel service. On entry to the close routine the device driver should remove the NDD_UP and NDD_RUNNING flags from the `ndd_flags` field.

ndd_close can be called from the process environment only.

A code fragment for an example Network Device Driver close routine follows:

```
xx_close(
  ndd_t       *p_ndd)    /* pointer to the ndd in the dev_ctl area */
{
  xx_dev_ctl_t   *p_dev_ctl = (xx_dev_ctl_t *)(p_ndd->ndd_correlator);
  int ipri;
 if (p_dev_ctl->device_state == OPENED) {
        p_dev_ctl->device_state = CLOSE_PENDING;
        /* wait for the transmit queue to drain */
        while (p_dev_ctl->device_state == CLOSE_PENDING &&
                (p_dev_ctl->tx_pending || p_dev_ctl->txq_len)) {
                DELAYMS(1000);          /* delay 1 second */
        }

  }

  /*
   * deactivate the adapter
   */
  p_dev_ctl->device_state = CLOSED;
  p_ndd->ndd_flags &= ~(NDD_RUNNING | NDD_UP | NDD_LIMBO | NDD_DEAD);

  unlock_enable(ipri, &SLIH_LOCK);

 /* cleanup all the resources allocated for open */
  ......
  unpincode(xx_open);
  return(0);
}
```

ndd_output Entry Point

This entry point has the following format:

```
ndd_output(struct ndd *ndd, struct mbuf *data)
```

The entry point has the following parameters:

- `ndd` is a pointer to the system **ndd** structure for this network device.
- `data` is a pointer to the mbuf chain to be transmitted.

DLPI or the socket network interface layer calls **ndd_output** *directly* to output data on the network device.

The first mbuf in each packet chain will be of the M_PKTHDR format (see /**usr/include/sys/mbuf.h**). Multiple mbufs may hold the packet and will be linked to data via the `m_next` field. Multiple packets on the transmit are supported and these will be linked to data via the `m_nextpkt` field. `m_pkthdr.len` is set equal to the total length of the individual packet.

On successful transmit, the device driver is responsible for freeing all mbufs. On failure, this is the callers responsibility. In the case of a multiple packet transmission, if any of the packets are transmitted the device driver should return success and free the mbufs.

On output, the device driver should check the value of `ndd->ndd_trace`. If this entry point is not null then the device driver should call the trace point for each packet which is transmitted (chained packets must be unchained before this call). The format of the call is:

```
(*ndd_trace) (struct ndd *ndd, struct mbuf *data,  caddr_t *hp,
ndd->ndd_trace_arg)
```

Parameters for **ndd_trace** have the following meanings:

- `ndd` is a pointer to the system ndd structure for this network device.
- `data` is a pointer to the mbuf chain to be transmitted.
- `hp` is a pointer to the start of the data in the mbuf being transmitted
- `ndd_trace_arg` is a cookie set by the trace routine.

The trace routine does not free the mbuf chain.

If transmission fails because of queue overruns, the device driver should return EAGAIN.

ndd_output may be called from the process or interrupt environment.

A code fragment for an example Network Device Driver output routine follows:

```
xx_output(
  ndd_t              *p_ndd,       /* pointer to the ndd in the dev_ctl area */
  struct mbuf        *p_mbuf)      /* pointer to a mbuf (chain) */
{
  xx_dev_ctl_t   *p_dev_ctl = (xx_dev_ctl_t *)(p_ndd->ndd_correlator);
  struct mbuf *p_cur_mbuf;
  struct mbuf *buf_tofree;
  int bus;
  int first;
  if (p_dev_ctl->device_state != OPENED) {
             return(ENETDOWN);
  }
  /*
   * if there is a transmit queue, put the packet onto the queue.
   */
  if (p_dev_ctl->txq_first) {
       /*
        *  if the txq is full, return EAGAIN. Otherwise, queue as
        *  many packets onto the transmit queue and free the
        *  rest of the packets, return no error.
        */
       .......
       ......
```

```
  }
  while (p_mbuf) {
       p_cur_mbuf = p_mbuf;
       /*
        * If there is txd available, try to transmit the packet.
        */
       if (!(WRK.txd_avail->flags & XX_IN_USE)) {

             if (!xx_xmit(p_dev_ctl, p_cur_mbuf, bus)) {
                 /*
                  * Transmit OK, free the packet
                  */
                 p_mbuf = p_mbuf->m_nextpkt;
                 p_cur_mbuf->m_nextpkt = NULL;;
                 m_freem(p_cur_mbuf);
                 first = FALSE;
             }
             else {
                 /*
                  * Transmit error. Call hardware error recovery
                  * function. If this is the first packet,
                  * return error. Otherwise, free the reset packets
                  * and return error.
                  */
                 p_ndd->ndd_genstats.ndd_oerrors++;
                 if (first) {

                    return(ENETDOWN);
                 }
                 else {
                    /* increment the error counter */
                    while (p_cur_mbuf = p_mbuf) {
                           p_mbuf = p_mbuf->m_nextpkt;
                           p_cur_mbuf->m_nextpkt = NULL;
                           m_freem(p_cur_mbuf);
                    }
                    return(0);
                 }
             }
       }    /* if there is txd available */
       else {
             while (p_cur_mbuf = buf_tofree) {
                    p_ndd->ndd_genstats.ndd_xmitque_ovf++;
                    p_ndd->ndd_genstats.ndd_opackets_drop++;
                    buf_tofree = buf_tofree->m_nextpkt;
                    p_cur_mbuf->m_nextpkt = NULL;
                    m_freem(p_cur_mbuf);
             }
             return(0);
       }
  }    /* while */
  return(0);
}

xx_xmit(
  xx_dev_ctl_t  *p_dev_ctl,/* pointer to the device control area */
  struct mbuf*p_mbuf,      /* pointer to the packet in mbuf */
  int         bus)         /* handle for I/O bus accessing */
{
       ndd_t   *p_ndd = &(p_dev_ctl->ndd);
       int count;
       int offset;
       int rc;
       int pio_rc = 0;
       /*
        * Call ndd_trace if it is enabled
        */
       if (p_ndd->ndd_trace) {
             (*(p_ndd->ndd_trace))(p_ndd, p_mbuf,
                    p_mbuf->m_data, p_ndd->ndd_trace_arg);
       }
```

```
        /* increment the tx_pending count */
        p_dev_ctl->tx_pending++;
        /*
         * copy data into transmit buffer and do processor cache flush
          */
         m_copydata(p_mbuf, 0, count, p_txd->buf);
        /*
         * Pad short packet with garbage
         */
        if (count < XX_MIN_MTU)
               count = XX_MIN_MTU;
        p_txd->tx_len = count;
        ......
        .....
        /*
         * tell the adapter how many bytes to send
         * clear the status and set the EOP and EL bit.
          */
         ......
         /* start watchdog timer */
         w_start(&(TXWDT));
        return(0);
}
```

ndd_ctl Entry Point

This entry point for the IOCTL system call has the following format:

```
ndd_ctl(struct ndd *ndd, int cmd, caddr_t arg, int length)
```

This entry point has the following parameters:

- `ndd` is a pointer to the system ndd structure for this network device.
- `cmd` is the IOCTL.
- `arg` is the address of the ioctl arguments. This address will be in kernel memory.
- `length` is the length of arg.

The ioctls for a network device driver are shown below:

NDD_ADD_FILTER
: Add receive data filter. `arg` points to the address of the filter function to add.

NDD_DEL_FILTER
: Remove receive data filter. `arg` points to the address of the filter function to remove.

NDD_ADD_STATUS
: Add status filter. `arg` points to the address of the filter function to add.

NDD_DEL_STATUS
: Delete status filter. `arg` points to the address of the filter function to delete.

NDD_CLEAR_STATS
: Clear all statistics maintained by the network device driver.

NDD_DISABLE_ADDRESS
: Disable a multicast address. Remove the NDD_ALTADDRS flag in the **ndd** structure. `arg` contains the multicast address.

NDD_ENABLE_ADDRESS
: Enable a multicast address. Set the NDD_ALTADDRS flag in the **ndd** structure. `arg` contains the multicast address.

NDD_GET_STATS
: Get general statistics from the network device. General statistics

are maintained by the device driver in the `ndd_genstats` field of the **ndd**. `arg` points to a user buffer where the `ndd_genstats` information should be placed.

NDD_GET_ALL_STATS
: Get all statistics from the network device. All statistics means general statistics maintained by the device driver in the `ndd_genstats` field of the **ndd** and additional specific statistics maintained by the device driver in a structure pointed to by the `ndd_specstats` field. Some device drivers may only update specific statistics when IOCTL commands are processed. `arg` points to a device-specific structure that contains the **ndd_genstats** structure followed by device-specific information.

NDD_MIB_QUERY
: Return the MIB structure identifying which options the device supports. `arg` is a pointer to a structure of type `generic_mib_t` in the kernel address space.

NDD_MIB_GET
: Get device specific MIBs. `arg` is a pointer to a structure of type `generic_mib_t` in the kernel address space.

NDD_DUMP_ADDR
: Return the address of remote dump routine in `arg`.

NDD_MIB_ADDR
: Get all receive addresses for the device. `arg` is a pointer to a structure of type `ndd_mib_addr_elem_t` in the kernel address space.

NDD_ENABLE_MULTICAST
: Enable receipt of all multicast packets. Set NDD_MULTICAST flag in **ndd** structure.

NDD_DISABLE_MULTICAST
: Disable receipt of all multicast packets. Remove NDD_MULTICAST flag in **ndd** structure.

NDD_PROMISCUOUS_ON
: Set promiscuous mode on if the adapter allows this. Promiscuous mode allows the adapter to receive all the packets transmitted on the network. Set NDD_PROMISC flag in **ndd** structure.

NDD_PROMISCUOUS_OFF
: Set promiscuous mode off. Remove NDD_PROMISC flag in **ndd** structure.

Network device drivers are not required to support all of the preceding IOCTLs. For IOCTLs not supported, the driver should return EOPNOTSUPP.

Additional device specific IOCTLs may also be supported.

DLPI or the socket network interface layer calls **ndd_ctl** *directly* if one of the preceding IOCTLs is issued on a stream or a raw (AF_NDD) socket.

ndd_ctl can be called from the process or interrupt environment.

Device Driver – CDLI Interface

This consists of the following kernel services and functions:

- **ns_attach** and **ns_detach**
- **nd_receive**
- **nd_status**

ns_attach and ns_detach Kernel Services

These have the following format:

```
ns_attach(struct ndd *ndd)
ns_detach(struct ndd *ndd)
```

The parameter `ndd` is a pointer to the system **ndd** structure for this network device.

The preceding two kernel services are called by network device driver configuration functions to add or remove their network device from the system's list of available devices. These services are discussed, in detail, in the initialization and CDLI–Device Driver Interface sections.

ns_attach and **ns_detach** can be called from the process environment only.

nd_receive Function

Network device drivers pass receive data to the system through calls to the **nd_receive** function that is specified in the **ndd** structure for the network device. This function may be null. The format of the call is:

The format of the call is:

```
(*nd_receive(struct ndd *ndd, struct mbuf *data))
```

The parameters are as follows:

- `ndd` is a pointer to the system **ndd** structure for this network device.
- `data` is a pointer to the **mbuf** chain to be received.

AIX supports multiple protocols, concurrently, on the same network device for both sockets and STREAMS users. To accomplish this receive packets are passed to network demuxers. Network demuxers are loadable kernel extensions whose function is to route packets to the appropriate user. Generally, a unique demuxer will be required for each type of network device. This is because knowledge of the physical layer data headers is required for packet routing. Network device driver writers will need to provide a network demuxer with their device driver if one of the system-provided demuxers is not satisfactory for their network device. Because both device drivers and network demuxers are loadable kernel extensions it is possible to add a completely new network device to AIX Version 4.1 without requiring any modifications to the base operating system. The device driver can either bind in a network demuxer or have the system add a demuxer appropriate for the type of network device being added. When DLPI or the socket network interface layer calls **ns_alloc** to register use of a network device driver, the system checks if the `ndd_demuxer` field in the **ndd** structure is null. If the field is null (the device driver has not bound in a demuxer) then the system searches the known demuxers in the system trying to find a demuxer for this type of network device. The interface type is set in the `ndd_type` field of the **ndd** structure and describes classes of network devices which have the same physical layer characteristics such as 802.3 Ethernet and Token Ring. All devices of the same type can use a common demuxer. If a match is found, **ns_alloc** sets the **ndd** receive and status function pointers to the demuxers. It also sets the `ndd_demuxsource` field to 0 (system provided). AIX Version 4.1 provides demuxers for the following interface types: NDD_ISO88023, NDD_ISO88025 and NDD_FDDI (see **/usr/include/sys/ndd.h**). Device driver writers for network devices of these types may elect to use the system demuxers instead of providing their own.

Typically, the **nd_receive** function is called by the device driver's receive interrupt routine. It is only called for receive frames that are not in error for any reason. The **mbuf** pointer passed to the **nd_receive** function may contain more than one **mbuf** chain.

nd_receive may be called with interrupts disabled.

nd_status Function

Network device drivers pass status event information to the system through calls to the **nd_status** function that is specified in the **ndd** structure. This function may be null.

The format of the call is:

```
(*nd_status(struct ndd *ndd, struct ndd_statblk *status))
```

The call has the following parameters:

- `ndd` is a pointer to the system **ndd** structure for this network device.
- `status` is a pointer to the status block. This structure is defined in **/usr/include/sys/ndd.h**

The **nd_status** function is typically called by the device driver's receive interrupt routine upon receipt of a bad frame. If the adapter has support for status interrupts, it should be called by the status interrupt routine. The status structure of type `ndd_statblk` should have the code field set to NDD_BAD_PKTS and values set in the option fields before **nd_status** is called. After calling the **nd_status** function, the interrupt routine should free the mbufs associated with the bad frame, or frames.

nd_status can be called from the process or interrupt environment.

Writing a Network Demuxer

Network demuxers, including the system provided ISO88023, ISO88025 and FDDI demuxers, are implemented as loadable kernel extensions in AIX Version 4.1 The following general guidelines apply:

- By convention, network demuxer kernel extensions are installed in the **/usr/lib/drivers** directory. Some system utilities may assume that this is the case.
- When building the kernel extension the relevant base system exports must be imported. The system provided demuxers import the following: **kernex.exp**, **syscalls.exp**. The system export files are located in the **/usr/lib** directory.
- Network demuxer writers must decide how much of the kernel extension to pin. The major consideration is that the demuxer's receive and status functions will be called with interrupts disabled. In general, the system provided demuxers are pinned in their entirety.
- Network demuxers must be configured and loaded into the system.

Network demuxers are loadable kernel extensions whose function is to route incoming packets to the appropriate user. Generally, a unique demuxer will be required for each type of network device. This is because knowledge of the physical layer data headers is required for packet routing. DLPI and the socket network interface layer make direct calls to network demuxers to register particular packet types to be received. Device drivers call network demuxers to route packets and status events to the correct users.

Demuxer Initialization

Network demuxer initialization involves execution of the kernel extension's configuration entry point. This configuration entry point is called by the configuration method of the network device driver during the system boot up. This function is invoked when the network demuxer is built and is called by the system when the kernel extension is loaded.The format of the entry should be as follows:

```
(*network_demuxer_config) (int cmd, struct uio *uio)
```

The entry has the following parameters:

`cmd`	Designates a particular demuxer command.
`uio`	Pointer to a **uio** structure. This is generally ignored by demuxers.

Command values that should be recognized by the network demuxer are:

CFG_INIT Initialize the demuxer.

CFG_TERM Terminate the demuxer.

Other demuxer specific commands can be defined.

When called with the CFG_INIT command the demuxer's configuration function should perform the following tasks:

- Do any pinning of modules or data structures required by the demuxer.
- Do any lock initialization required by the demuxer.
- Initalize the demuxer control structures.
- Call the **ns_add_demux** kernel service to add the demuxer to the system's list of available demuxers. This step is not required if the demuxer is to be bound to a device driver or is not to be made available to other network devices.

When called with the CFG_TERM command the demuxer's configuration function should perform the following tasks:

- Verify that there are no active users of the demuxer.
- Do any unpinning of modules or data structures required by the demuxer.
- Free any lock resources.
- Free any demuxer control structures.
- If required, call the **ns_del_demux** kernel service to remove the demuxer from the system's list of available demuxers.

nd_add_filter Function

The **nd_add_filter** function adds a receive filter for the routing of received data. It is invoked by the CDLI routines and has the following entry point:

```
(*nd_add_filter)(struct ndd *nddp, caddr_t filter, int len, ns_user_t
ns_user)
```

The entry point has the following parameters:

`nddp` Pointer to the system **ndd** structure for this demuxer.

`filter` Address of a user defined structure which contains information that the demuxer needs such as filter type. See /**usr**/**include**/**sys**/**cdli.h** for the **ns_8022** structure as an example of what is used for this parameter with 802.2 networks.

`len` Length in bytes of the filter parameter.

`ns_user` Pointer to the **ns_user** structure that describes the user of the filter. The structure is defined in /**usr**/**include**/**sys**/**cdli.h**.

The `nd_add_filter` field of the **ns_demuxer** structure filter is set to the address of this function during configuration of the demuxer. The function should perform the following tasks:

- Set any necessary data structure locks.
- Perform sanity checks on the fields of the **ns_user** structure.
- Handle the appropriate filter type specified in the filter parameter, if applicable.
- Set the `nddp->ndd_specdemux` field to a pointer to a private demuxer control structure that maintains a list of which filters have been added. Be sure to save **ns_user** information here; it will later be retrieved by the **nd_receive** function.

- Call the **nddp–>ndd_ctl** entry point of the Network Device Driver to register the filter with the NDD. The calling format is:

```
((*nddp->ndd_ctl))(nddp,NDD_ADD_FILTER,filter,len);
```

- If any errors are returned from the preceding call, invoke the appropriate demuxer delete filter function.
- Release all locks set upon entry into the function.
- Return 0 for success, **errno** otherwise.

nd_del_filter Function

The **nd_del_filter** function for the demuxer deletes a previously specified receive filter. It is invoked by the CDLI routines and has the following entry point:

```
(*nd_del_filter)(struct ndd *nddp, caddr_t filter, int len)
```

The entry point has the following parameters:

`nddp`	Pointer to the system **ndd** structure for this demuxer.
`filter`	Address of a user defined structure which contains information about the filter that is to be deleted.
`len`	Length in bytes of the filter parameter.

The `nd_del_filter` field of the **ns_demuxer** structure filter is set to the address of this function during configuration of the demuxer. The function should perform the following tasks:

- Set any necessary data structure locks.
- Check the `len` parameter for correctness and verify that a filter has previously been added.
- If applicable, handle the appropriate filter type specified in the filter parameter.
- Retrieve the filter list from `nddp->ndd_specdemux`, search for the filter to be deleted and free any storage associated with that filter.
- Call the **nddp–>ndd_ctl** entry point of the Network Device Driver to inform it that the filter is being deleted. On return from this entry point, check for nonzero error return codes. The calling format is:

```
((*nddp->ndd_ctl))(nddp,NDD_DEL_FILTER,filter,len);
```

- Release all locks set upon entry into the function.
- Return 0 for success, **errno** otherwise.

nd_add_status Function

The **nd_add_status** function adds a filter for routing asynchronous status. It is invoked by the CDLI routines and has the following entry point:

```
(*nd_add_status)(struct ndd *nddp, caddr_t filter, int len, ns_statuser *ns_statuser)
```

The entry point has the following parameters:

`nddp`	Pointer to the system **ndd** structure for this demuxer.
`filter`	Address of a user defined structure which contains information that the demuxer needs such as filter type. See **/usr/include/sys/cdli.h** for the **ns_com_status** structure as an example of what is typically used for this parameter.
`len`	Length in bytes of the filter parameter.
`ns_statuser`	Pointer to structure describing the status user. The structure is defined in **/usr/include/sys/cdli.h**.

The `nd_add_status` field of the **ns_demuxer** structure filter is set to the address of this function during configuration of the demuxer. The function should perform the following tasks:

- Set any necessary data structure locks.
- Check the `len` parameter for correctness and verify that a filter has previously been added.
- Call the **dmx_add_status** kernel service to add the status filter. The calling format is:

```
dmx_add_status(nddp, filter, ns_statuser)
```

- If no errors were encountered, call the **nddp–>ndd_ctl** entry point of the Network Device Driver to register the status filter. On return from this entry point, check for nonzero error return codes. The calling format is:

```
((*nddp->ndd_ctl))(nddp,NDD_ADD_STATUS,filter,len);
```

- Release all locks set upon entry into the function.
- Return 0 for success, **errno** otherwise.

nd_del_status Function

The **nd_del_status** function deletes a previously added status filter. It is invoked by the CDLI routines and has the following entry point:

```
(*nd_del_status)(struct ndd *nddp, caddr_t filter, int len)
```

The parameters are the same as the parameters of the companion **nd_add_status** function.

The `nd_del_status` field of the **ns_demuxer** structure filter is set to the address of this function during configuration of the demuxer. The function should perform the following tasks:

- Set any necessary data structure locks.
- Check the `len` parameter for correctness and verify that a filter has previously been added.

Call the **dmx_del_status** kernel service to delete the status filter. The calling format is:

```
dmx_del_status(nddp, filter)
```

- If no errors were encountered, call the **nddp–>ndd_ctl** entry point of the Network Device Driver to unregister the status filter. On return from this entry point, check for nonzero error return codes. The calling format is:

```
((*nddp->ndd_ctl))(nddp,NDD_DEL_STATUS,filter,len);
```

- Release all locks set upon entry into the function.
- Return 0 for success, **errno** otherwise.

nd_receive Function

The **nd_receive** function is invoked by the Network Device Driver for receive packets.

The `nd_receive` field of the **ns_demuxer** structure filter is set to the address of this function during configuration of the demuxer. The function should perform the following tasks:

- Set any necessary data structure locks.
- If there are no receive filters, free all the mbufs and return.
- Grab each mbuf in the chain and parse the header data to determine which type of additional processing needs to be done. For example, an Ethernet driver will send the **nd_receive** function 802.3 packets, or standard Ethernet type packets, or both of these types of packets. Use the macros DELIVER_PACKET or IFSTUFF_AND_DELIVER found in **/usr/include/net/nd_lan.h** to send the data to the next layer. For 802.2 style networks, the dmx_8022_receive kernel service has been provided to send the data to the next layer. The entry point has the following format:

  ```
  dmx_8022_receive(struct ndd *nddp, struct mbuf *m, int len)
  ```

  For this entry point, the *len* parameter is the size of the MAC header in bytes.
- After exhausting all the mbufs in the chain, release all locks and return. The return value for this function is void.

For more information, see the description of **nd_receive**, on page 13-11, in "Writing a Network Device Driver."

nd_status Function

The **nd_status** function is invoked by the Network Device Driver for all status conditions. It distributes asynchronous status to the appropriate network services users.

The `nd_status` field of the **ns_demuxer** structure filter is set to the address of this function during configuration of the demuxer. The function should perform the following tasks:

- Set any necessary data structure locks.
- Pass status information to the next layer by calling the **dmx_status** kernel service. The calling format is:

  ```
  dmx_status(struct ndd *nddp, struct ndd_statblk *status)
  ```
- Release all locks set upon entry into the function. The return value of this function is void.

For more information, see the description of **nd_status**, on page 13-12, in "Writing a Network Device Driver."

nd_response Function

The **nd_response** function is invoked by the higher level protocols if they choose to perform 802.2 LLC Exchange Station ID or TEST Link Frame processing. It has the following entry point:

```
(*nd_response)(struct ndd *nddp, struct mbuf *m, int llcoffset)
```

The entry point has the following parameters:

`ndd`	Pointer to the system **ndd** structure for this demuxer.
`m`	Poiner to a mbuf structure which may not contain more than one packet which contains a XID or TEST packet.
`llcoffset`	Byte offset to the start of the `llc` header in the mbuf.

The `nd_response` field of the **ns_demuxer** structure filter is set to the address of this function during configuration of the demuxer. The function should perform the following tasks:

- Set any necessary data structure locks.
- Copy the MAC source address for the MAC destination address and copy the `nddp->ndd_physaddr` to the MAC source address.
- If the MAC destination address indicates routing information is present, turn off the routing control bits in the MAC header.
- Call the device driver output routine `(*nddp->ndd_output)(nddp,m)` to send the packet to the driver. If a nonzero error is returned, free the mbuf.
- Release all locks set upon entry into the function. The return value of this function is void.

DLPI/Socket – Network Demuxer Interface

When a network demuxer is bound to a network device driver either by the device driver or through a **ns_alloc** call (**ns_alloc** is described in the appendix of this book), a linkage is created between the **ndd** structure and the demuxer's **ns_demuxer** structure (defined in **/usr/include/sys/cdli.h**). This is done by setting the `ndd_demuxer` field in the **ndd** structure equal to the address of the demuxer's structure. When DLPI or the socket network interface issues a request to the system (using **ns_add_filter** or **ns_del_filter**) to start receiving or stop receiving packets or status on this network interface, the system invokes the demuxer's entry points for adding or deleting filters or status as defined in the **ns_demuxer** structure. A sample fragment of code for **ns_add_filter** is:

```
/*****************************************************************
*
*
*   ns_add_filter() -  Pass "add filter" request on to demuxer
*
*****************************************************************
/
ns_add_filter(nddp, filter, len, ns_user)
     struct ndd     *nddp;     /* specific interface  */
     caddr_t        filter;
     int           len;
     struct ns_user  *ns_user;  /* the details  */
{
      return((*(nddp->ndd_demuxer->nd_add_filter))
                  (nddp, filter, len, ns_user));
}
```

Thus these kernel service are really entry points into the network demuxer. The syntax of the calls are as follows:

```
ns_add_filter(struct ndd *ndd, caddr_t  filter, int len,
                struct ns_user *ns_user);
```

The following parameters apply to **ns_add_filter**:

`ndd`	Pointer to the system **ndd** structure for this network device. Typically, a demuxer user obtains this pointer through a prior call to **ns_alloc**.
`filter`	Address of the filter function to be added. This is demuxer dependent.
`len`	Length of the filter.
`ns_user`	Pointer to a **ns_user** structure. This structure is defined in **/usr/include/sys/cdli.h**.

"Sample Code – DLPI Call to ns_add_filter," on page 13-20, shows how DLPI calls **ns_add_filter**. First, a **ns_alloc** call is made to obtain the **ndd** pointer. DLPI builds the interface name from the basename of the device being opened (for example, "**tr**" for /**dev**/**dlpi**/**tr**) and the physical point of attachment (for example, 0) passed by the user in the DL_ATTACH_REQ. Next the sample illustrates how DLPI handles a DL_SUBS_BIND_REQ request to receive packets from this interface that match a specific 802.2 logical link control (LLC) header. Note that DLPI simply passes the users request intact to the demuxer. Also note how the **ns_user** structure is built. The `isr` field points to the DLPI interrupt handler and `isr_data` is set to the internal address of this Stream. When the demuxer calls DLPI on a receive packet it also returns `isr_data` and DLPI does not need to do **any** packet demultiplexing to locate the target user. The `netisr` field is NULL indicating that the DLPI interrupt routine is to be called on the interrupt level. `pkt_format` tells the demuxer how much of the LLC header to remove on input (and how much to expect on output). DLPI simply passes the format which the user has set through an IOCTL.

ns_add_status has the following syntax:

```
ns_add_status(struct ndd *ndd, caddr_t  filter, int len, struct
ns_statuser *ns_statuser);
```

The following parameters apply to **ns_add_status**:

`ndd`	Pointer to the system ndd structure for this network device. Typically, a demuxer user obtains this pointer through a prior call to **ns_alloc**.
`filter`	Address of the status function to be added. This is demuxer dependent.
`len`	Length of the filter.
`ns_statuser`	Pointer to a **ns_statuser** structure. This structure is defined in /**usr**/**include**/**sys**/**cdli.h**.

When **ns_add_status** is called, the demuxer should issue a **ndd_ctl** with the NDD_ADD_STATUS operation.

The call to reverse **ns_add_filter** is **ns_del_filter**. The call to reverse **ns_add_status** is **ns_del_status**.

ns_del_filter has the following syntax:

```
ns_del_filter(struct ndd *ndd, caddr_t filter, int len)
```

The function arguments for **ns_del_filter** are the same as those for **ns_add_filter**.

ns_del_status has the following syntax:

```
ns_del_status(struct ndd *ndd, caddr_t filter, int len)
```

The function arguments for **ns_del_status** are the same as those for **ns_del_filte**r.

The demuxer should issue a **ndd_ctl** with the NDD_DEL_STATUS operation when the **ns_del_status** entry point is called.

Many of the demuxer functions for 802.2 LANs can be handled by common routines. A demuxer can register to use these functions by setting `nd_use_nsdmx` in the demuxers **ns_demuxer** structure to the true value. Setting `nd_use_nsdmx` true causes the **dmx_init** function to be called when **ns_alloc** binds a network demuxer to a system **ndd** structure.

Device Driver – Network Demuxer Interface

The Network Device Driver determines which demuxer to use by setting the `ndd_type` field during configuration in its config entry point routine. (For more information see “Writing a Network Device Driver”, on page 13-2.) A list of defined `ndd_types` can be found in /**usr**/**include**/**sys**/**ndd.h**; a typical name is NDD_ISO88023. The command **ifconfig** calls the configuration method of the NDD, which calls the configuration method for the demuxer. In addition, **ifconfig** calls the configuration method for the associated NID for the interface. (For more information, see “Loading and Initialization” in “Network Interface Driver Functions,” on page 13-21.)

The demuxer’s configuration entry point passes to the **ns_add_demux** function a structure of type **ns_demuxer** that contains fields that are set to the address of functions that the CDLI and Network Device Driver routines eventually call. For the NDD, this means that the `nd_receive` and `nd_status` fields of the **ndd** structure are used. These fields contain function pointers to the appropriate demuxer functions. (For more information, see the discussion of the **nd_receive** function, on page 13-16, and the discussion of the **nd_status** function, on page 13-16.)

Sample Code – DLPI Call to ns_add_filter

```
/*
 * dlb_attach - attach interface to this module
 *
 */
staticf MBLKP
dlb_attach(dlb, mp)
      DLBP  dlb;
      MBLKP mp;
{
      int len;
      int ppa;
      char *name;
      NDDP ndd;
...
      ppa = ((dl_attach_req_t *)(mp->b_rptr))->dl_ppa;
      name = mknddname(dlb->dlb_nddname, ppa);
      if (ns_alloc(name, &ndd)) {
...
      }
...
}
/*
 * dlb_subs_bind - bind a SNAP
 *
 * if dlb_isap != 0xAA, then either
 *    - we bound to some non-snap sap
 *    - we have already done a subs_bind
 */staticf MBLKP
dlb_subs_bind(dlb, mp)
      DLBP  dlb;
      MBLKP mp;
{
      dl_subs_bind_req_t *dlsbr = (dl_subs_bind_req_t *)mp->b_rptr;
      snap_t *snap = (snap_t *)(mp->b_rptr +
dlsbr->dl_subs_sap_offset);
      int err = 0;
      struct      ns_8022     dl;
      struct      ns_user     ns_user;
...
      else if (dlsbr->dl_subs_bind_class != DL_HIERARCHICAL_BIND)
            err = DL_NOTSUPPORTED;
      else if (dlsbr->dl_subs_sap_length != sizeof(snap_t))
            err = DL_BADADDR;
      else if ((char *)snap + sizeof(snap_t) > mp->b_wptr)
            err = DL_BADADDR;
...
      ns_user.isr = (int)dlb_intr;
      ns_user.isr_data = (caddr_t)dlb;
      ns_user.protoq = nilp(struct ifqueue);
      ns_user.netisr = NULL;
      ns_user.ifp = nilp(struct ifnet);
      ns_user.pkt_format = dlb->dlb_pkt_format;
      dl.filtertype = NS_8022_LLC_DSAP_SNAP;
      dl.dsap = dlb->dlb_isap;
      bcopy(snap, dl.orgcode, sizeof(snap_t));
      if (err = ns_add_filter(dlb->dlb_ndd, &dl, sizeof(dl), &ns_user))

            return dlb_error(mp, DL_SUBS_BIND_REQ, err, dlb);
...
}
```

Writing a Network Interface Driver

The AIX Network Interface Driver (NID) is a layer of software between a network device driver (NDD) and an AIX network layer. This layer is required for all network device drivers that have to be made available to a network layer. This discussion concentrates on NIDs for an Internet Protocol (IP) network layer, though you can easily modify an NID to support other network layers.

An AIX Network Interface Driver provides a uniform interface to the IP layer. The NID passes output packets from the IP layer to the Network Device Driver (NDD). The device driver insulates the NID from the hardware but does not hide the special anomaly from the type of underlying physical network.

Each type of physical network has unique access requirement. For Ethernet, this is the Ethernet header. For a Token Ring, it is the MAC/LLC header. These different anomalies of the network are handled inside the NID, providing a uniform interface to the network protocols.

Basic Functions of a Network Interface Driver

The NID is a dynamically loadable kernel extension similar to a device driver. You must load or add the NID to the kernel through a configuration method. A typical NID performs the following basic functions:

- Provides a uniform interface from the network layer to the network device driver.
- Translates an IP address to a hardware address for the underlaying device driver.
- Builds the communication device driver specific protocol header (see the Data Packet for Ethernet figure, on page 13-25).
- Communicates with the network device driver.

A specific NID may perform more functions than those previously listed.

Summary of NID Changes in AIX Version 4.1

Several functions which were in AIX Version 3.2 NIDs have been removed from the NIDs and placed into the Network Demuxer. For example, the Receive Data and Status Interrupt function has been moved to the Network Demuxer. (For more information on this, see “Writing a Network Demuxer”, on page 13-12.) The **add_arp_iftype**, **del_arp_iftype**, and **find_arp_iftype** kernel services are no longer available, making it the responsibility of NID writers to either supply their own ARP and address resolution routines or to use the one supplied by the system. There is an all new set of NID specific IOCTL calls. Finally, there are some small changes regarding initialization and termination of the NID.

Network Interface Driver Functions

A network interface driver (NID) performs the following functions:

- Loading and initializing
- Communicating with the NDD
- Translating network addresses to hardware addresses
- Handling NID specific IOCTL calls
- Terminating and unloading

Loading and Initialization

The configuration method, **ifconfig**, loads the AIX NID kernel extension.

The **ifconfig** command configures the correct NID with the correct Network Device Driver (NDD). The non-numeric portion of the interface parameter from the command line is used as the NDD name. The NID name is the same as the NDD name with `if_` prepended to

the NDD name. For example, the following command tells **ifconfig** to configure NID **if_en** and NDD **en** in /**usr**/**lib**/**drivers**:

```
ifconfig en0 1.1.1.1 up
```

The **ifconfig** command is run automatically when the system is started, and configures all NIDs that have been defined in the ODM database. For your NID to be automatically configured, your interface must be defined in the CuDv ODM object class with the parent attribute set to `inet0` and the name attribute set to the name of your interface. In addition, the CuAt ODM object class must have a corresponding netaddr attribute defined for the interface.

If you choose not to have **ifconfig** configure your NID automatically, you can manually issue the command as shown above, or place the command in one of the shell scripts that are run when the system is started.

After loading the NID, the configuration method calls the NID kernel extension entry point with the **CFG_INIT** command. This initializes the AIX NID.

Steps during initialization include:

- Pin the code and data for the NID kernel extension if not already pinned. The code might be pinned already if the configuration method for this NID has been invoked earlier for another adapter of the same type. If there are multiple adapters of the same type, the same NID code services all these adapters. It is not necessary to load multiple copies of an NID for multiple adapters of the same type.
- Open and initialize the underlying NDD. This is done by the **ns_alloc** subroutine in the following sample code.
- Initialize an **ifnet** structure and call the **if_attach** kernel service. This is done by **xx_attach** in the following sample code. The **if_attach** kernel service adds a NID to the network interface list, which is a linked list of **ifnet** structures.

  Initialize the **ifnet** structure with the address of the **output** routine (ifp–>if_output), **ioctl** routine (ifp–>if_ioctl), and **reset** routine (ifp–>if_reset).

The following sample code illustrates loading and initializing:

```
struct  xx_softc {
      struct  arpcom xx_ac;   /* common part         */
      struct ndd      *nddp;
};

config_xx(cmd, uio)
int         cmd;
struct uio  *uio;
{
      struct      device_req        device;
      int                     error = 0;
      int                     unit;
      struct      xx_softc          *xxp;
      char                    *cp;
      int                     type;
      if ( (cmd != CFG_INIT) || (uio == NULL) )
            return(EINVAL);
      if (uiomove((caddr_t) &device, (int)sizeof(device), UIO_WRITE, uio))
            return(EFAULT);
      lockl(&if_xx_lock, LOCK_SHORT);
      if (init == 0) {
            if (ifsize <= 0)
                  ifsize = IF_SIZE;
            if (error = pincode(config_xx))
                  goto out;
            xx_softc = (struct xx_softc *)
                  xmalloc(sizeof(struct xx_softc)*ifsize, 2, pinned_heap);
```

```
            if ( (xx_softc == (struct xx_softc *)NULL) ||
                 (xx_softc == (struct xx_softc *)NULL) ) {
                  unpincode(config_xx);
                  unlockl(&if_xx_lock);
                  return(ENOMEM);
            }
            bzero(xx_softc, sizeof(struct xx_softc) * ifsize);
            init++;
      }
      cp = device.dev_name;
      while(*cp < '0' || *cp > '9') cp++;
      unit = atoi(cp);
      if (unit >= ifsize) {
            error = ENXIO;
            goto out;
      }
      if (!strncmp(device.dev_name, "xx", 2)) {
            type = IFT_XX;
            xxp = &xx_softc[unit];
      } else {
            error = EINVAL;
            goto out;
      }
      error = ns_alloc(device.dev_name, &xxp->nddp);
      if (error == 0)
            xx_attach(unit, type);
      else
            bsdlog(LOG_ERR,
                  "if_xx: ns_alloc(%s) failed with errno = %d\n",
                  device.dev_name, error);
out:
      unlockl(&if_xx_lock);
      return(error);
}
xx_attach(unit, type)
unsigned          unit;
unsigned          type;
{
      register struct ifnet         *ifp;
      register struct xx_softc        *xxp;
      extern int              xx_output();
      extern int              xx_ioctl();
      xxp             = &xx_softc[unit];
      ifp             = &xxp->xx_if
      ifp->if_name    = "xx";
      ifp->if_mtu     = ??;
      ifp->if_type    = type;
      ifp->if_unit    = unit;
      bcopy(xxp->nddp->ndd_physaddr, xxp->xx_addr, 6);
      ifp->if_flags     = IFF_BROADCAST | IFF_NOTRAILERS;
      /* Check if the adapter supports local echo */
      if (xxp->nddp->ndd_flags & NDD_SIMPLEX)
            ifp->if_flags     |= IFF_SIMPLEX;
      ifp->if_output    = xx_output;
      ifp->if_ioctl     = xx_ioctl;
      ifp->if_addrlen = ??;
      ifp->if_hdrlen  = ??;
      ifp->if_mtu     = ??;
      ifp->if_unit    = unit;
      ifp->if_name    = "xx";
      ifp->if_init    = xx_init;
      ifp->if_output  = xx_output;
      ifp->if_ioctl   = xx_ioctl;
      ifp->if_type    = IFT_XX;
```

```
        ifp->if_addrlen = ??;
        ifp->if_hdrlen  = ??;
        /* packet filter support */
        ifp->if_flags     |= IFF_BPF;   /* Enable bpf support */
        ifp->if_tap = NULL;       /* Inactive tap filter */
        ifp->if_arpres = arpresolve;
        ifp->if_arprev = revarpinput;
        ifp->if_arpinput = arpinput;
        if_attach(ifp);
}
```

Communicating with the IP

The TCP/IP and NIDs are different kernel extensions that are loaded separately. In order to communicate with each other, these kernel extensions use the functions and data structures of the base kernel extension. These kernel services include:

- Address Family Domain kernel services
- Network Interface Device kernel services
- Routing and Interface Address kernel services

Some of these services are:

add_input_type
Adds an interface type to the Network Input Table.

del_input_type
Deletes an input type from the Network Input Table.

find_input_type
Finds an input type from the Network Input Table.

if_attach Adds a network interface to the network interface list**.**

if_detach Deletes a network interface from the network interface list.

ifunit Returns the **ifnet** structure for the requested interface.

ifa_ifwithaddr Locates an interface based on a complete interface 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 Changes statistical elements of the interface array to zero in preparation for an attach operation.

Outgoing Packets

For the outgoing packets, the routing code in the IP layer determines the correct NID to use by scanning the linked list of **ifnet** structures. The routing code uses Interface Address kernel services like **ifa_ifwithaddr, ifa_ifwithdstaddr**, and **ifa_ifwithnet** to locate the appropriate **ifnet** structure. For more detailed information on kernel services, see the *AIX Technical Reference, Volume 5: Kernel and Subsystems*.

The **ifnet** structure is initialized by each NID during its initialization phase. Once the appropriate NID is located, the IP layer calls the **output** routine of the NID:

```
(*ifp->if_output)(ifp, m, dst, rt)
        struct ifnet *ifp;
        struct mbuf  *m;
        struct sockaddr *dst;
        struct rtentry *rt;
```

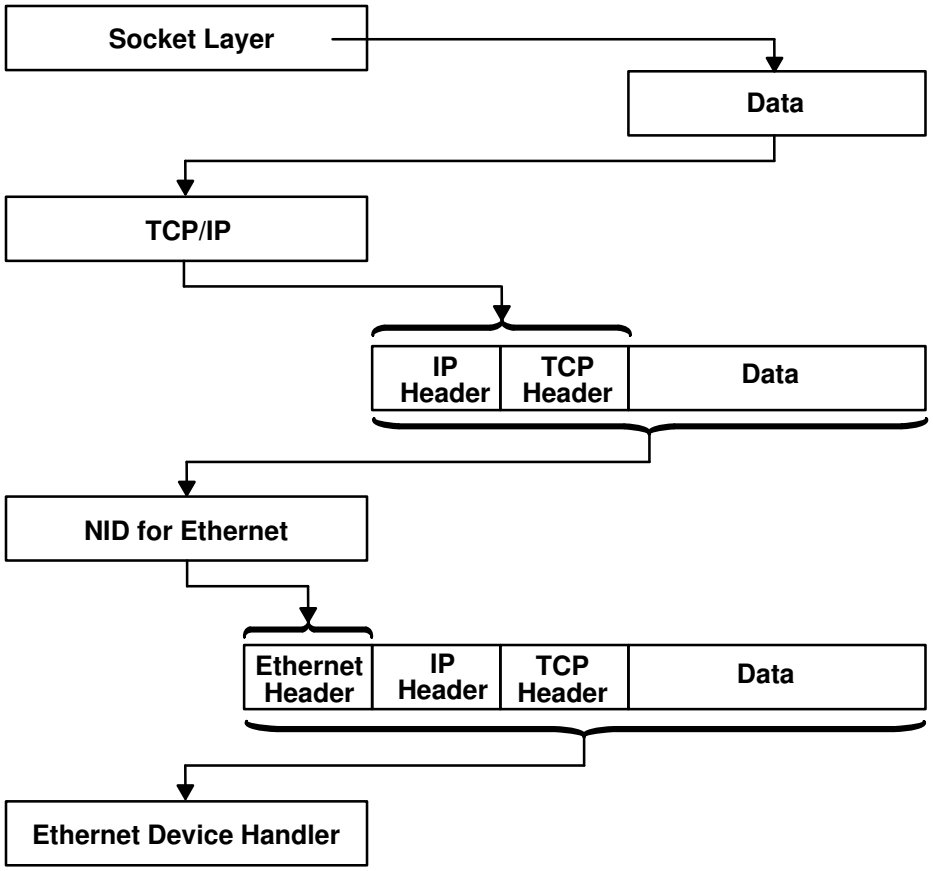
ifp A pointer to the **ifnet** structure. This is required as there may be multiple adapters of the same type that are serviced by the same NID. The `if_unit` field in the **ifnet** structure is used to determine the appropriate adapter.

m The mbuf chain containing the data packet. This mbuf chain is freed by the NID or the device handler.

dst A pointer to the socket address of the destination of the packet.

rt A pointer to a routing table entry for this packet. A null value indicates there is no routing table entry.

The following Data Packet for Ethernet figure illustrates the modifications to an outgoing packet from the socket layer to the Ethernet device handler.



Data Packet for Ethernet

The **output** routine provides no guarantee for the transmission of packets. There is no acknowledgment of a successful delivery. The errors returned are those that can be detected immediately, such as the interface is down, no memory buffers, and address family not supported. If the error is detected after the call is returned, the protocol is *not* notified.

The following sample code shows typical code for an **output** routine:

```
/*****************************************************************************
 *      xx_output() - output packet to network
 *****************************************************************************/
xx_output(ifp, m, dst, rt)
register struct ifnet *ifp;
register struct mbuf *m;
struct sockaddr_xx *dst;
struct rtentry *rt;
{
        register struct xx_softc                *xxp;
        register struct xx_hdr                  *hdrp;
        struct xx_hdr                           hdr;
        register int                            hdr_len;
        register int                            error = 0;
        struct mbuf                             *mcopy = 0;

        if ((ifp->if_flags & (IFF_UP|IFF_RUNNING)) != (IFF_UP|IFF_RUNNING))
{
                error = ENETDOWN;
                ++xxp->if_snd.ifq_drops;
                goto out;
        }

        if ((xxp->if_flags & IFF_SIMPLEX) && (m->m_flags & M_BCAST))
                mcopy = m_copym(m, 0, (int)M_COPYALL, M_DONTWAIT);

        xxp = &xx_softc[xxp->if_unit];

        switch (dst->sa_family) {
                /*
                  call necessary ARP resolve routine or any other resolve
                  here
                */
                hdrp = (struct xx_hdr *)&hdr;
                break;
        }

        /*
         * Add local net header.  If no space in first mbuf,
         * allocate another.
         */
        hdr_len = xx_len;
        M_PREPEND(m, hdr_len, M_DONTWAIT);
        if (!m) {
                error = ENOBUFS;
                ++xxp->if_snd.ifq_drops;
                goto out;
        }

        /* copy in the local net  headers */
        bcopy((caddr_t)hdrp, mtod(m, caddr_t), hdr_len);

        m = m_collapse(m, xx_MAX_GATHERS);
        if (!m) {
                error = ENOBUFS;
                ++xxp->if_snd.ifq_drops;
                goto out;
        }

        if (m->m_flags & M_BCAST|M_MCAST)
                ++xxp->if_omcasts;
```

```
        xxp->if_obytes += m->m_pkthdr.len;
        if (!xxp->nddp) {
                m_freem(m);
                ++xxp->if_oerrors;
        /*
           call the Network Device Driver's output function
        */
        } else if ((*(xxp->nddp->ndd_output))(xxp->nddp, m)) {
                m_freem(m);
                ++xxp->if_oerrors;
        }
        if (mcopy)
                (void) looutput(xxp, mcopy, dst, rt);
        ++xxp->if_opackets;
        m = 0;
    out:
        if (m)
                m_freem(m);
        return (error);
}
```

Communicating with the Device Handler

The communications device handler interface kernel services provide a standard interface between NIDs and AIX communication device handlers.

The **ns_alloc** and **ns_free** services allocate and relinquish use of a Network Device Driver, respectively. Once the NDD has been allocated, outgoing packets to the NDD can be sent by calling the driver's output function found in the `ndd_output` field of the **ndd** structure. This structure is returned after a successful call to **ns_alloc**.

Note: **ns_alloc** and **ns_free** are described in the appendix of this book.

Output Data

The **output** routine for NID is called by the network layer (IP). The NID transmits the data by calling the function found in `ndd_output` of the `ndd` structure. This is shown in the **xx_output** routine in the sample code starting on page 13-26.

Before calling the **ndd_output** function, the output routine might have to build the corresponding link-level header. If the **output** routine is called by IP, it is supplied with a destination address in a **sockaddr** structure. If the sockaddr address family is supported by the NID, the NID has to map this sockaddr into a link-level address.

This mapping may involve a lookup, or it may require more involved techniques like an Address Resolution Protocol (ARP). For more information on ARPs, see "Translating Network Addresses to Hardware Addresses," on page 13-28.

It may not be always necessary to build the link-level header. For example, a point-to-point link may not need a link-level header.

The **ndd_output** function pointer requires two parameters. The first is a pointer to the `ndd` structure of which `ndd_output` is a field. The second is a pointer to the `mbuf` with contains the data to transmit. The first `mbuf` in each packet chain will be of the `M_PKTHDR` format (see **/usr/include/sys/mbuf.h**) Multiple `mbuf`s may hold the packet and will be linked to data the `m_next` field. **ndd_output** points to a device driver entry point similar to the following:

```
xx_output(p_ndd, p_mbuf)
ndd_t       *p_ndd;  /* pointer to the ndd in the dev_ctl area */
struct mbuf *p_mbuf; /* pointer to a mbuf (chain) */
```

This function returns a zero after successful transmission. If a nonzero return is encountered, it is the NIDs responsibility to free the mbufs.

Translating Network Addresses to Hardware Addresses

The network layer provides the NID with the destination network address. If it is not a point-to-point network the NID must translate this network address into a destination hardware address to perform a successful transmission of the packet.

For the existing AIX NIDs, different mechanisms are used for different types of NIDs. For Ethernet, Token Ring, 802.3, and Fiber Distributed Data Interface (FDDI), the Address Resolution Protocol (ARP) is used.

The ARP is defined in the RFC826. The ARP provides a dynamic address-translation mechanism for networks that support broadcast or multicast communication.

The idea of ARP is simple. Whenever a packet needs to be sent out, the NID calls the ARP resolver routine to get the hardware address of the destination. If the address is already known (in the cache translation table) then that value is returned. If not, the packet is queued and an ARP request is broadcast on the network. The request has the network address of the required destination host.

When the correct destination host receives the ARP request, it sends back an ARP reply providing the requester with the hardware destination address. The original sender host can then update its cache translation table and can transmit the queued-up output packet.

The resolver routines for the previously mentioned networks (Ethernet, Token Ring, 802.3, and FDDI) for the Internet address family are provided by the corresponding NID kernel extension.

The resolver routine is specified by the NID during configuration. The `if_arpres` field of the `ifnet` structure is assigned a pointer to the NID specific ARP resolution routine before calling **if_attach** kernel service. The resolver routine will then be called by the IP layer at the appropriate time by looking at the `ifnet` structure for that particular interface.

The **resolve** routine resolves the IP address into the corresponding hardware address. If successful, *addr_hw* points to the hardware address, and a value of zero is returned to the caller.

If there is no entry in the ARP table (arptab), one needs to be created. An entry is created with the network address of **sock_addr_net** and a broadcast ARP request is sent out (using **xx_arpwhohas**). The mbuf structure containing the data is held until the address is resolved. A value of 1 is returned to the caller. The response for the ARP request is eventually received by the **xx_arpinput** routine. This routine would update the ARP table and send out the held packet.

```
(*resolve)(ac, m, sock_addr_net, addr_hw)
struct arpcom *ac;
struct mbuf *m;
struct in_addr *sock_addr_net;
caddr_t *addr_hw;
```

Based on the interface type of the resolver, the *addr_hw* parameter has different semantics:

Ethernet *addr_hw* is used as type `struct ether_header *eh`. It is used to return the value of the Ethernet address.

Token-ring (802.5)
addr_hw is used as type `struct ie5_mac_hdr *macp`. It completes the MAC and LCC headers.

802.3 *addr_hw* is used as type `struct ie3_mac_hdr *macp`. It completes the MAC and LCC headers.

FDDI *addr_hw* is used as type `struct fddi_mac_hdr *macp`. It completes the MAC and LCC headers.

The ARP input routine is specified by the NID during configuration. The `if_arpinput` field of the `ifnet` structure is assigned a pointer to the NID specific ARP resolution routine

before calling the **if_attach** kernel service. The ARP input routine will then be called by the receive function of the interface demuxer at the interrupt level.

For Local Area Networks, several resolver and arp input routines are present with the NIDs provided with the system. These NIDs, in /**usr**/**lib**/**drivers**, are for the following interfaces:

if_en Standard Ethernet and IEEE 802.3 Ethernet NID

if_fd FDDI NID

if_tr IEEE 802.5 token-ring NID

If you choose to use the arp routines provided with the system instead of writing your own, use the following values for the *if_arpres* and i*f_arpinput* fields of the **ifnet** structure:

Interface Type	if_arpres Value	if_arpinput Value
Standard Ethernet	arpresolve	arpinput
FDDI	fddi_arpresolve	fddi_arpinput
802.5	ie5_arpresolve	ie5_arpinput
802.3	arpresolve	arpinput

The **whohas** routine broadcasts an ARP request packet asking who has the sock_addr_net Internet Address.

```
(*whohas)(ac, sock_addr_net)
struct arpcom *ac;
struct in_addr *sock_addr_net;
```

The **arptfree** routine frees any ARP table entry specified by `*ac`. Any mbuf chain of data associated with this arptab entry and held for future transmission, is also freed. (See the previous discussion of the **resolve** routine, about one page earlier, in this same section.)

```
(*arptfree)(at)
struct arptab *at;
```

Handling NID Specific ioctl Calls

The NID supports at least the following **ioctl** commands:

SIOCSIFADDR Sets the Network Interface address.

The SIOCIFADDR command does not update the interface address list. The address list is updated by the network (for example, TCP/IP) layer. See the **ifaddr** structure on page 13-35 for more information.

The SIOCIFADDR only adds the IP address to the **arpcom** structure for the address in the AF_INET domain.

SIOCSIFFLAGS Sets interface flags.

The SIOCSIFFLAGS command modifies the `if_flags` field in the **ifnet** structure for this NID. The different flags that are stored in the `if_flags` field are as follows (also see the **net/if.h** header file):

SIOCIFDETACH Calls the **ns_free** function to deallocate the given device driver.

IFIOCTL_ADD_FILTER Calls the **ns_add_filter** CDLI service to add the given filter to the device driver

IFIOCTL_DEL_FILTER Calls the **ns_del_filter** CDLI service to remove the given filter

SIOCADDMULTI Adds a multicast address. The IOCTL must support subfunctions for adding a multicast address and for enabling a multicast address.

SIOCDELMULTI Deletes a multicast address. The IOCTL must support subfunctions for deleting a multicast address and for disabling all multicasts.

The following sample code shows an IOCTL routine:

```
xx_ioctl(ifp, cmd, data)
         register struct ifnet              *ifp;
         int                                cmd;
         caddr_t                            data;
{
         register struct ifaddr             *ifa = (struct ifaddr *)data;
         register struct xx_softc   *xxp = &xx_softc[ifp->if_unit];
         int                                error = 0;
         struct timestruc_t                 ct;

         if (!xxp->nddp) {
                  return(ENODEV);
         }

         switch (cmd) {
             case SIOCIFDETACH:
                  ns_free(xxp->nddp);
                  xxp->nddp = 0;
                  break;

             case IFIOCTL_ADD_FILTER:
                  {
                  ns_8022_t         *filter;
                  ns_user_t         *user;

                  filter = &((struct if_filter *)data)->filter;
                  user = &((struct if_filter *)data)->user;
                  error = ns_add_filter(xxp->nddp, filter, sizeof(*filter),
                           user);
                  if (error)
                           bsdlog(LOG_ERR,
                              "if_fd: ns_add_filter() failed with %d return
code.\n",
                                 error);
                  break;
                  }

             case IFIOCTL_DEL_FILTER:
                  ns_del_filter(xxp->nddp, (ns_8022_t *)data, sizeof(ns_8022_t));
                  break;

             case SIOCSIFADDR:
                switch (ifa->ifa_addr->sa_family) {
                     case AF_INET:
                          ((struct arpcom *)ifp)->ac_ipaddr =
                                   IA_SIN(ifa)->sin_addr;
                          break;

                     default:
                          break;
                  }
                  fd_init();
                  ifp->if_flags |= IFF_UP;
                  curtime(&ct);
                  ifp->if_lastchange.tv_sec = (int)ct.tv_sec;
                  ifp->if_lastchange.tv_usec = (int)ct.tv_nsec / 1000;
                  break;

             case SIOCSIFFLAGS:
                  fd_init();
                  break;

             case SIOCADDMULTI:
                  {
                  register struct ifreq *ifr = (struct ifreq *)data;
                  void xx_map_ip_multicast();
                  char     addr[6];

                  /*
```

```
                 * Update our multicast list.
                 */
                switch(driver_addmulti(ifr, &(xxp->fd_ac),
                         xx_map_ip_multicast, &error, addr)) {
                    case ADD_ADDRESS:
                         error = (*(xxp->nddp->ndd_ctl))
                                      (xxp->nddp, NDD_ENABLE_ADDRESS, addr,
                                              FDDI_ADDRLEN);
                         if(error) {
                                 int rc;
                                 driver_delmulti(ifr, &(xxp->fd_ac),
                                      xx_map_ip_multicast, &rc, addr);
                         }
                         break;

                    case ENABLE_ALL_MULTICASTS:
                         error = (*(xxp->nddp->ndd_ctl))
                                      (xxp->nddp, NDD_ENABLE_MULTICAST, addr,
                                              FDDI_ADDRLEN);
                         if(error) {
                                 int rc;
                                 driver_delmulti(ifr, &(xxp->fd_ac),
                                      xx_map_ip_multicast, &rc, addr);
                         }
                         break;

                    case 0:
                         /* address already enabled */
                         break;
                    case -1:
                         /* error */
                         break;
                }
                break;
                }

            case SIOCDELMULTI:
                {
                register struct ifreq *ifr = (struct ifreq *)data;
                void xx_map_ip_multicast();
                char addr[6];
                /*
                 * Update our multicast list.
                 */
                switch(driver_delmulti(ifr, &(xxp->fd_ac),
                         xx_map_ip_multicast, &error, addr))
                {
                    case DEL_ADDRESS:
                        error = (*(xxp->nddp->ndd_ctl))
                                 (xxp->nddp, NDD_DISABLE_ADDRESS, addr,
                                          FDDI_ADDRLEN);
                        break;

                   case DISABLE_ALL_MULTICASTS:
                         error = (*(xxp->nddp->ndd_ctl))
                                 (xxp->nddp, NDD_DISABLE_MULTICAST, addr,
                                          FDDI_ADDRLEN);
                         break;
                    case 0:
                         /* address still in use */
                         break;
                    case -1:
                         /* error */
                         break;
                 }
                 break;
              }

            default:
                 error = EINVAL;
        }
```

```
        return (error);
}
```

Two kernel services have been added to help with multicast addressing. They are **driver_addmulti** to add multicast addresses and **driver_delmulti** to delete multicast addresses.

```
driver_addmulti(ifr, *ac, func, error, mac_address)
struct ifreq *ifr;
struct arpcom *ac;
void func();
int *error;
char *mac_address;
```

The `func` parameter above is a pointer to a user defined routine in the NID that maps the multicast address.The following is an example of such a routine:

```
void xx_map_ip_multicast(struct sockaddr_in *sin, u_char *lo,
u_char *hi)
{
if (sin->sin_addr.s_addr == INADDR_ANY) {
        /*
        * An IP address of INADDR_ANY means listen to all
        * of the xx multicast addresses used for IP.
        * (This is for the sake of IP multicast routers.)
        */
        bcopy(xx_ipmulticast_min, lo, 6);
        bcopy(xx_ipmulticast_max, hi, 6);
} else {
        xx_MAP_IP_MULTICAST(&sin->sin_addr, lo);
        bcopy(lo, hi, 6);
}
}
```

The function **driver_delmulti** has the same parameters.

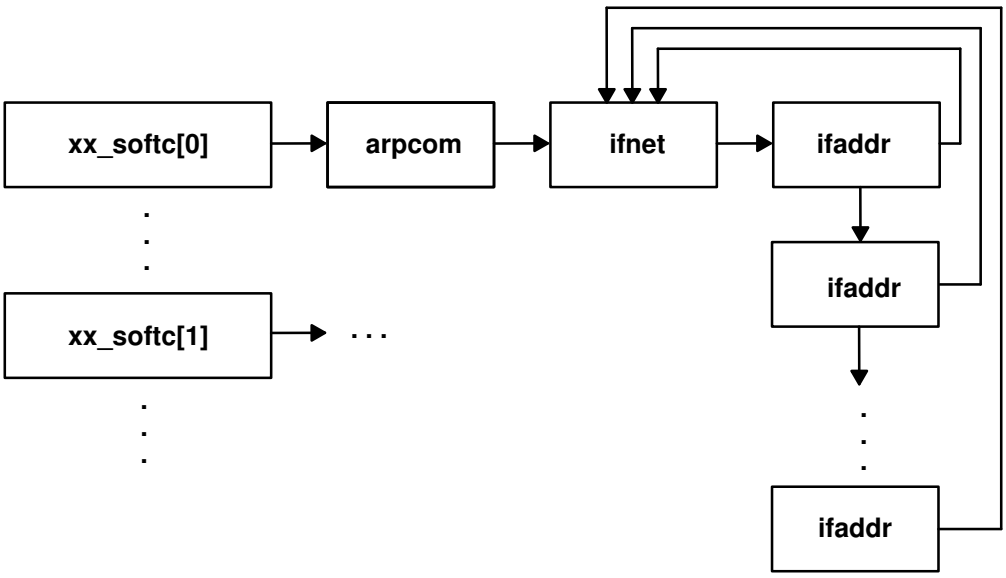
```
driver_delmulti(ifr, *ac, func, error, mac_address)
struct ifreq *ifr;
struct arpcom *ac;
void func();
int *error;
char *mac_address;
```

Terminating

The IOCTL SIOCIFDETACH discussed in “Handling NID Specific IOCTL Calls,” on page 13-29, supports the termination of the NDD and corresponding NID.

NID and ARP Data Structures

This section lists data structures used for NID and ARP. The following NID Data Structure Relationships figure shows the relationships between various NID and ARP data structures.



NID Data Structure Relationships

xx_softc

The **xx_softc** structure shows *software status* and is an internal structure of the NID. This is a starting point for locating the address of other data structures for the NID.

```
struct  xx_softc {
    struct arpcom  xx_ac;  /* common part                  */
    struct  ndd    *nddp;  /* ptr to device driver struct */
}
```

arpcom

The **arpcom** structure is a *network common* structure. It is shared between the NID and the address resolution code. This is the first element in the **softc** structure.

```
struct arpcom {
   struct ifnet ac_if;
   u_char ac_hwaddr[MAX_HWADDR];
   struct in_addr ac_ipaddr;
   struct driver_mult *ac_multiaddrs; /* Lock for walking list */
                                      /* of multicast addrs.   */
};
```

ifnet

The **ifnet** structure is the Network Interface Table. It has the following types of fields:

- Interface identifier (if_name, ...)
- Interface properties (if_mtu, if_flags, ...)
- Interface routines (if_output, if_ioctl, ...)
- Interface statistics (if_ipackets, if_opackets, if_ierrors, ...)

It also maintains a pointer to the linked list interface addresses (if_addrlist) for the interface. The **ifnet** structures for the different interfaces are in a linked list (if_next). The following code sample shows the **ifnet** structure:

```
 struct ifnet {
        char    *if_name;               /* name, e.g. ``en'' or ``lo'' */
        short   if_unit;                /* sub-unit for lower level driver */
        u_long  if_mtu;                 /* maximum transmission unit */
        u_long  if_flags;               /* up/down, broadcast, etc. */
        short   if_timer;               /* time 'til if_watchdog called */
        int     if_metric;              /* routing metric (external only) */
        struct  ifaddr *if_addrlist;    /* linked list of addresses per if */
/* procedure handles */
        int     (*if_init)();           /* init routine */
        int     (*if_output)();         /* output routine (enqueue) */
        int     (*if_start)();          /* initiate output routine */
        int     (*if_done)();           /* output complete routine */
        int     (*if_ioctl)();          /* ioctl routine */
        int     (*if_reset)();          /* bus reset routine */
        int     (*if_watchdog)();       /* timer routine */
/* generic interface statistics */
        int     if_ipackets;            /* packets received on interface */
        int     if_ierrors;             /* input errors on interface */
        int     if_opackets;            /* packets sent on interface */
        int     if_oerrors;             /* output errors on interface */
        int     if_collisions;          /* collisions on csma interfaces */
/* end statistics */
        struct  ifnet *if_next;
        u_char  if_type;                /* ethernet, tokenring, etc */
        u_char  if_addrlen;             /* media address length */
        u_char  if_hdrlen;              /* media header length */
        u_char  if_index;               /* numeric abbreviation for this if  */
/* SNMP statistics */
        struct  timeval if_lastchange;  /* last updated */
        int     if_ibytes;              /* total number of octets received */
        int     if_obytes;              /* total number of octets sent */
        int     if_imcasts;             /* packets received via multicast */
        int     if_omcasts;             /* packets sent via multicast */
        int     if_iqdrops;             /* dropped on input, this interface */
        int     if_noproto;             /* destined for unsupported protocol */
        int     if_baudrate;            /* linespeed */


/* stuff for device driver */
        dev_t   devno;                  /* device number */
        chan_t  chan;                   /* channel of mpx device */
        struct  in_multi *if_multiaddrs; /* list of multicast addresses */
        int     (*if_tap)();            /* packet tap */
        caddr_t if_tapctl;              /* link for tap (ie BPF) */
        int     (*if_arpres)();         /* arp resolver routine */
        int     (*if_arprev)();         /* Reverse-ARP input routine */
        int     (*if_arpinput)();       /* arp input routine */
        struct  ifqueue {
                struct  mbuf *ifq_head;
                struct  mbuf *ifq_tail;
                int     ifq_len;
                int     ifq_maxlen;
                int     ifq_drops;
        } if_snd;                       /* output queue */
        simple_lock_data_t      if_slock;       /* statistics lock */
        simple_lock_data_t      if_multi_lock;
};
```

ifaddr

The **ifaddr** structure contains information about one interface address. The structures are maintained by the different address families (for example, Internet, osi, and xns). They are allocated and attached by the address families, and *not* by the NID. All the addresses for an interface are linked so that they are easy to locate.

```
struct ifaddr {
        struct  sockaddr *ifa_addr;     /* address of interface */
        struct  sockaddr *ifa_dstaddr;  /* other end of p-to-p link */
#define ifa_broadaddr   ifa_dstaddr     /* broadcast address interface */
        struct  sockaddr *ifa_netmask;  /* used to determine subnet */
        struct  ifnet *ifa_ifp;         /* back-pointer to interface */
        struct  ifaddr *ifa_next;       /* next address for interface */
#ifdef  _KERNEL
        void    (*ifa_rtrequest)(int, struct rtentry *, struct sockaddr *);
#else
        void    (*ifa_rtrequest)();     /* check or clean routes (+ or -)'d
*/
#endif
        struct  rtentry *ifa_rt;        /* ??? for ROUTETOIF */
        u_short ifa_flags;              /* mostly rt_flags for cloning */
        u_short ifa_llinfolen;          /* extra to malloc for link info */
};
```

ifreq

The **ifreq** (interface request) structure is used for socket IOCTLs. All interface IOCTLs must have parameter definitions that begin with `ifr_name`. The remainder can be interface-specific.

```
struct  ifreq {
#define IFNAMSIZ        16
        char    ifr_name[IFNAMSIZ];     /* if name, e.g. "en0" */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                struct  sockaddr ifru_broadaddr;
                long    ifru_flags;
                int     ifru_metric;
                caddr_t ifru_data;
                short   ifru_mtu;
        }
```

arptab

The **arptab** (ARP table entries) structures contain the network address to link-layer address translation. The table is maintained by the ARP routines.

The entries in this table are hashed for fast retrieval. The table is divided into ARPTAB_NB buckets, each of size ARPTAB_BSIZ. The network address entry to be stored is hashed based upon its *at_iaddr* value into the appropriate bucket. To work on this table, ARPTAB_HASH and ARPTAB_LOOK are defined.

```
struct  arptab {
        struct in_addr  at_iaddr;/* internet address */
        u_char          hwaddr[MAX_HWADDR];/* hardware address */
        u_char          at_timer;/* minutes since last reference */
        u_char          at_flags;/* flags */
        struct mbuf     *at_hold;/* last pkt til resolved/timeout
*/
        struct ifnet    *at_ifp;/* ifnet assoc with entry */
        union if_dependent if_dependent;/* hdwr dependent info */
};
```

arpreq

Use the **arpreq** (ARP request from user) structure to initiate a socket ioctl request.

```
struct  arpreq {
        struct sockaddr     arp_pa;/* protocol address */
        struct sockaddr     arp_ha;/* hardware address */
        int                 arp_flags;/* flags */
        u_short             at_length;/* length of hdwr addr */
        union if_dependent  ifd;/* hdwr dependent info */
        u_long              ifType;/* interface type */
}
```

Include Files

The include files for the data structures include:

net/if.h
net/if_arp.h
sys/mbuf.h
sys/socket.h
sys/devinfo.h

Tracing and Debugging for NIDs

The **trace** tool is useful for debugging the NID kernel extension. It is also a useful mechanism for performance analysis and tuning. The tracing code has a small overhead, so the system performance is minimally altered by using tracing code. To add **trace** code, see the chapter on the trace facility in *AIX General Programming Concepts : Writing and Debugging Programs* and “Performance Tracing” on page 15-86. The process of tracing is described in the following paragraphs.

Each logical module inside the kernel and kernel extension is allocated a unique hook ID. For example:

- HKWD_SOCKET for the socket module
- HKWD_MBUF for the mbuf module
- HKWD_IFEN for the Ethernet NID.
- HKWD_IFFD for the FDDI NID
- HKWD_IFET for the Ethernet 802.3 NID
- HKWD_IFTR for the Token Ring 802.5 NID
- HKWD_IFSL for the SLIP NID

For each of the hook IDs, there are sub-hook IDs for the procedures contained in the related modules.

For example, for the Ethernet NID output routine, the sub-hook IDs are `hkwd_output_in` and `hkwd_output_out`. A new kernel extension can use tracing by allocating itself a new unique hook ID which does *not* have the same value as an existing hook ID. A combination of hook ID and sub hook ID is used to create a unique trace event in the trace log.

For example, at the beginning of the Ethernet ARP resolve routine, there is the tracing call:

```
TRCHKT(HKWD_IFEN | hkwd_output_in)
```

Start tracing by using the **trace** command. Stop it by calling **trcstop**. Process the trace log by using the **trcrpt** command.

The **net_error** kernel service makes a tracing call to trace the error generated.

```
TRCHKL1(HKWD_NETERR | error_code, ifp)
```

Configuration Method for NID

Since NID is a dynamically loadable kernel extension, it has to be loaded by an application which is called as a configuration method. For the existing NID of AIX this is achieved using the **ifconfig** command.

To add your own NID used by AIX TCP/IP, create a configuration method that loads your own NID. This configuration method will be similar to **ifconfig**.

The configuration method uses the **sysconfig** subroutine to load, unload, and configure the AIX NID.

Chapter 14. Network Interfaces and Protocols

AIX Version 4.1 provides two standard user application interfaces to the network: sockets and STREAMS. The figure CDLI Device Driver Structure, on page 13-1, graphically displays the network interface architecture.

Network device drivers are not required to support the socket and STREAMS interfaces directly. Instead device drivers interface with kernel services which provide the upper layer support. Collectively these kernel services are called the Common Data Link Interface (CDLI). Nor are device drivers required to support direct user access to the driver. Raw socket and STREAMS interfaces are provided for this purpose. Because device drivers only interface with well defined *kernel* services, this architecture greatly simplifies the writing and porting of network device drivers.

AIX Version 4.1 supports user written STREAMS and socket network protocols. For the STREAMS user, both a tli and an xti application library is provided along with the OSF/ 1.2 STREAMS system call framework. The xti application interface is X/Open compliant. STREAMS network protocols written to the TPI (Transport Provider Interface) should be able to directly link into the STREAMS application interface from below. A Data Link Provider Interface (DLPI) STREAMS module is provided to connect STREAMS-based protocols with network device drivers. In general, STREAMS protocols should not have to make direct calls to either CDLI or the network device driver. For socket users, AIX Version 4.1 offers a standard BSD socket framework with a set of kernel services which permit users to dynamically add both communication domains and new protocol switch entries to an existing communications domain, but this feature is currently supported only for the AF_INET address family. In addition, AIX exports a number of low level socket calls such as sbappend, sbdrop and sbreserve which are required by socket based protocols. With these services, user written socket based network protocols should be able to directly link into the socket application interface from below. The standard BSD network interface layer (ifnet) is provided to connect socket based protocols with network device drivers. In general, socket based protocols should not have to make direct calls to either CDLI or the network device driver.

Finally, both network device drivers and network protocols are implemented as loadable kernel extensions. Special commands, (such as **strload** for STREAMS) or special user written commands (similiar to **ifconfig** for sockets) accomplish the loading of these kernel extensions.

STREAMS User Interfaces

Both the Transport Layer Interface (TLI) and the X/Open Transport Interface (XTI) define a transport service interface. The TLI and XTI implementations are limited to TCP/IP. TLI and XTI are accessed through a set of library subroutines for the C programming language. These libraries access the kernel through STREAMS messages.

TLI and XTI provide both a connection mode and a connectionless mode of transport services. A transport endpoint specifies a communication path between the transport user and the transport provider. A single transport endpoint may not support both modes of service simultaneously.

TLI and XTI support both synchronous and asynchronous execution modes for handling asynchronous events. In synchronous mode, the user program is blocked until a specific event has occurred. In asynchronous mode, the user process is not blocked and is notified when the specific event has occurred.

Both TLI and XTI support a rich option management facility to provide negotiation, debugging, graceful shutdown, and buffer management functionalities.

The TLI implementation is based on AT&T SVR4 (System V release 4). The transport provider driver, XTISO, and the associated STREAMS module, timod, are based on the Transport Provider Interface (TPI) version 1.5. The XTI implementation complies with the XPG4 standards defined by X/Open.

The user can also directly call the STREAMS **getmsg** and **putmsg** system calls to read from and write to the stream-head message queue.

For more information see “xtiso STREAMS Driver” and “dlpi STREAMS Driver” in *AIX Technical Reference, Volume 4: Communications*, “Transport Service Library Interface Overview” in *AIX Communications Programming Concepts*, and the reference information on **getmsg** and **putmsg** in *AIX Technical Reference, Volume 4: Communications*.

Protocol Interfaces via DLPI

The DLS provider is implemented as a style 2 provider that supports both the connectionless and connection-oriented modes of communication. The DLPI services support local management primitives and data-transfer primitives. Local management primitives allow the DLS user to query and control the DLS provider. Data-transfer primitives enable the DLS user to communicate with a peer DLS user. Each local management primitive and data-transfer primitive is implemented as a **STREAMS** message. Normal primitives are implemented as **M_PROTO** messages. High-priority interface acknowledgement primitives are implemented as **M_PCPROTO** messages.

A style 2 DLS provider can support several physical points of attachment (PPA). The PPA is used to identify one of several of the same type of interface in the system. A DLS user must explicitly identify the PPA using the **DL_ATTACH_REQ** primitive. For more information, see the description of the **DL_ATTACH_REQ** primitive in *AIX Technical Reference, Volume 3: Communications*.

The DLS provider has been implemented to allow the DLS user the capability of specifying the packet format. Using the **M_IOCTL STREAMS** message, the DLS user can specify the packet format. If the DLS user does not specify the packet format, the default is **NS_PROTO**. The packet formats are defined in “DLPI Interfaces Supported by AIX,” on page 14-3.”

In order for the DLS provider to generically support all interface types, the DLS provider has been implemented so the DLS user can specify address resolution routines.

For more information, see the reference articles on the DLPI primitives (such as **DL_ATTACH_REQ**) in *AIX Technical Reference, Volume 3: Communications* and *AIX Technical Reference, Volume 4: Communications*.

See “Data Link Provider Interface Information” in *AIX Communications Programming Concepts* for more information on protocol interfaces using DLPI.

Writing or Porting STREAMS Network Protocols

This section:

- Lists and explains DLPI interfaces supported by AIX
- Details the AIX interpretations of source and destination address
- Provides sample code for packet format setting
- Points to demuxer docs

DLPI Interfaces Supported by AIX

The DLPI driver supports CDLI-based network interfaces in a generic fashion. This support is enabled by allowing the DLPI user to specify the particular packet format necessary for the transmission media over which the stream is created. Using the **M_IOCTL** streams message, the DLS user can specify the packet format. If the DLS user does not specify the packet format, the default is **NS_PROTO**. The packet formats are defined in **/usr/include/sys/cdli.h**. The definitions of these formats follow:

NS_PROTO Remove all link-level headers. SNAP is not used.

NS_PROTO_SNAP
Remove all link-level headers including SNAP.

NS_PROTO_DL_COMPAT
Use the AIX Version 3.2.5 DLPI address format.

NS_PROTO_DL_DONTCARE
No addresses present in **DL_UNITDATA_IND**.

NS_INCLUDE_LLC
Leave LLC headers in place.

NS_INCLUDE_MAC
Do not remove any headers.

The DLPI user is allowed one packet format specification per stream. This packet format must be specified after the attach, and before the bind. Otherwise, an error is generated. The DLPI user can specify one of the following packet formats per stream: **NS_PROTO**, **NS_PROTO_SNAP**, **NS_PROTO_DL_COMPAT**, **NS_PROTO_DL_DONTCARE**, **NS_INCLUDE_LLC**, and **NS_INCLUDE_MAC**. The **NS_PROTO**, **NS_PROTO_SNAP**, **NS_INCLUDE_LLC**, and **NS_INCLUDE_MAC** packet formats are defined in the **/usr/include/sys/cdli.h** file. The **NS_PROTO_DL_COMPAT** and **NS_PROTO_DL_DONTCARE** packet formats are defined in the **/usr/include/sys/dlpi_aix.h** file. If the user does not specify a packet format, the default packet format is **NS_PROTO**.

For the **DL_UNITDATA_IND** primitive, DLPI provides the header information in the **dl_unitdata_ind_t** structure. All packet formats except **NS_INCLUDE_MAC** accept downstream addresses in the form mac_addr.dsap[.snap].

If the packet format specified is **NS_PROTO** or **NS_PROTO_SNAP**, the MAC and LLC are included in the header, and the data portion of the message contains only data. If the packet format is **NS_PROTO**, the DLPI header includes the MAC and LLC without the SNAP. If the packet format is **NS_PROTO_SNAP**, the DLPI header includes the MAC, LLC, and SNAP. Both **NS_PROTO** and **NS_PROTO_SNAP** present destination addresses as mac_addr and source addresses as mac_addr.ssap.dsap.ctrl[.snap].

If the packet format specified is **NS_PROTO_DONTCARE**, the DLPI driver does not place any addresses in the upstream **DL_UNITDATA_IND**. Addresses are still required on the **DL_UNITDATA_REQ**.

If the packet format specified is **NLS_PROTO_DL_COMPAT**, DLPI uses the address format used in the AIX Version 3.2.5 DLPI driver. The address format is identical both upstream and downstream, and the source and destination addresses are presented as mac_addr.dsap[.snap].

If the packet format specified is **NS_INCLUDE_LLC**, the DLPI header contains only the destination and source addresses. If the packet format is **NS_INCLUDE_LLC**, only the LLC is placed in the data portion of the message. If the packet format is **NS_INCLUDE_MAC**, the MAC and LLC are both placed in the data portion of the message. Therefore, the DLPI user must have knowledge of the MAC header and LLC architecture for that interface in order to retrieve the MAC header and LLC from the data portion of the message. The **NS_INCLUDE_LLC** packet format presents both the source and destination addresses as mac_addr and leaves the LLC header in the **M_DATA** portion of the **DL_UNITDATA_IND** message. The **NS_INCLUDE_MAC** format sets the stream to raw mode, which does not process incoming or outgoing messages. (The previous connectionless-only DLPI driver sent messages upstream consisting of an **M_PROTO**/**DL_UNITDATA_IND** prepended to an **M_DATA** block containing the entire medium frame.) This verson of the driver does not prepend any **M_PROTO** portion, instead presenting only **M_DATA** messages upstream. This is because the received messages may not be Unitdata type, but any of the LLC message types. Downstream messages no longer require the **DL_UNITDATA_REQ** header and must be received as **M_DATA** messages. These messages must contain a completed MAC header that will be copied to the medium without further translation.

For the **DL_UNITDATA_REQ** primitive, if the DLPI user had specified either the **NS_PROTO**, **NS_PROTO_SNAP**, or **NS_INCLUDE_LLC** format, the DLPI user must provide the destination address and an optional DSAP in the DLPI header. If the DLPI user does not specify the DSAP, the DSAP specified at bind time is used. If the DLPI user specifies the **NS_INCLUDE_LLC** packet format, the user must include only the LLC in the data portion. If the user specifies the **NS_INCLUDE_MAC** packet format, the DLPI user must provide the full MAC header, including the LLC, in the data portion of the message.

The DLPI user specifies the packet format via the **STREAMS I_STR** IOCTL. For example, to change the address format to one compatible with AIX Version 3.2.5, enter:

```
int
fixaddr(int fd) {
      int old=NS_PROTO_DL_COMPAT;
      return ioctl(fd, DL_PKT_FORMAT, old);
}
```

To select a raw stream, enter:

```
int
beraw(int fd) {
      int raw+NS_INCLUDE_MAC;
      return ioctl(fd, DL_PKT_FORMAT, raw);
}
```

AIX Interpretations of Source and Destination Addresses

DLPI source and destination addresses are 6 byte hardware addresses.

TLI and XTI source and destination addresses are specified via the **sockaddr_in** structure defined in the /**usr**/**include**/**netinet**/**in.h** file. This structure requires a family, a port number, and an IP address.

Protocol Address Resolution

In order for the DLPI driver to generically support all interface types, DLPI has been implemented to allow the DLS user the capability of specifying address resolution routines.

The DLPI user can provide an address resolution procedure for input and output using the STREAMS **I_STR** IOCTL, or the user can rely on the system default address resolution routines. AIX provides default address resolution routines that are interface specific. The default input address resolution routine is **ndd**–>**nd_demuxer**–>**nd_address_input**, and the default output address resolution routine is **ndd**–>**ndd_demuxer**–> **nd_address_resolve**. (Refer to /**usr**/**include**/**sys**/**ndd.h**).

The DLPI driver calls the input address resolution routine with a pointer to the MAC header (and optional LLC header) and a pointer to an **mbuf** structure containing data. The contents of the data depend on which packet format was specified by the user.

The DLPI driver calls the output address resolution routine with a pointer to an **output_bundle** structure, an **mbuf** structure, and an **ndd** structure. The **output_bundle** structure is described in /**usr**/**include**/**net**/**nd_lan.h**. The DLPI driver assigns the destination address to key_to_find and copies the pkt_format and bind time LLC into helpers. If the user has provided a different dsap/type than what was set at bind time, the DLPI driver copies these values into helpers. It is the output resolution routine's responsibility to complete the MAC header and call **ndd_output( )**.

If you choose to specify the input and/or output address resolution routine, use the following sample code:

```
noinres(int fd) {
        return ioctl(fd, DL_INPUT_RESOLVE, 0);
}
```

AIX STREAMS Loading Convention

The configuration file for DLPI is the /**etc**/**dlpi.conf** file.

To load: strload –f /etc/dlpi.conf

To unload: strload –uf /etc/dlpi.conf

For STREAMS modules and drivers that can be accessed by TLI and XTI, the configuration file is the /**etc**/**xtiso.conf** file.

To load: strload –f /etc/xtiso.conf

To unload: strload –uf /etc/xtiso.conf

MP Serialization and Locking Options for STREAMS Modules and Drivers

The STREAMS framework has a special set of data structures and synchronizations that enable STREAMS-based devices to operate in a multi-threaded environment.

At configuration time, users supply valid synchronization levels for STREAMS modules and drivers. The synchronization level is specified in the sc_sqlevel member of the **strconf_t** structure passed in as an argument into the **str_install** configuration procedure call. In addition, MP-safe and MP-efficient STREAMS drivers and modules are required specify in the sc_flags member of the **strconf_t** structure the style of the open routine logically ORed with STR_MPSAFE.

Valid synchronization levels are:

SQLVL_QUEUE
: Queue Level. This synchronization level enables a separate thread of execution to access either side of the stream simultaneously.This synchronization level provides the finest degree of parallelization. If the user specifies the SQLVL_QUEUE level of synchronization, the user may need to provide locks to ensure that only one thread executes within a queue at a time. If the user must provide a locking mechanism, it is recommended that the user employ the q_lock.

SQLVL_QUEUEPAIR
: Queue Pair Level. This synchronization level guarantees that only one thread of execution can access either queue of the queue pair at a time.

SQLVL_MODULE
: Module Level. This level specifies synchronous across all instances of a module, ensuring no more than one thread of execution through all instances of the module at a time. This is the level of synchronization the user should select in a non MP-safe STREAMS module.

SQLVL_ELSEWHERE
: Arbitrary Level. A cooperating group of modules, such as a protocol family, may need to ensure that there is only one thread of execution through the entire group. In this case, the module developer provides a unique name that is used at configuration time. The unique name is specified in the sc_sqinfo member of the **strconf_t** structure.

SQLVL_GLOBAL
: Global Level. This synchronization level forces a single thread access through all streams. This option is normally used only for debugging. With this level of synchronization, the user requests a single lock for the entire streams system. Only one thread at a time may be executing.

SQLVL_DEFAULT
: Default Level. This synchronization level is defined as SQLVL_MODULE.

TLI and XTI Interface Protocols

The Transport Provider Interface (TPI) is a STREAMS message interface that specifies the types and allowable sequences of messages passed between the transport user and the transport provider. TPI is defined by a set of primitives which are implemented as STREAMS messages. These STREAMS messages can consist of M_PROTO, M_PCPROTO, or M_DATA message blocks. The TLI and XTI libraries are implemented using TPI primitives.

The TLI module, **timod**, sits between the stream head and the transport provider and helps map TLI messages to TPI primitives. **timod** passes most messages along unchanged.

Obtaining Copies of the DLPI Specifications

You can obtain copies of the Data Link Provider Interface (DLPI) specifications electronically. A postscript version of the DLPI specifications may be retrieved electronically by anonymous ftp from any of the internet hosts listed below.

HOST	IP address	Pathname
liasun3.epfl.ch	128.178.155.12	/pub/sun/dlpi
marsh.cs.curtin.edu.au	134.7.1.1	/pub/netman/dlpi
ftp.eu.net	192.16.202.2	/network/netman/dlpi
opcom.sun.ca	142.77.1.61	/pub/drivers/dlpi
ftp.cac.psu.edu	128.118.2.23	/pub/unix/netman/dlpi

To retrieve the postscript DLPI specifications through anonymous ftp, use the following example:

```
ftp ftp.eu.net
Connected to eunet.EU.net.
220-
220-Welcome to the central EUnet Archive,
220-
220 eunet.EU.net FTP server (Version wu-2.4(2) Jul 09 1993) ready.
Name (ftp.eu.net:jhaug):anonymous
ftp> user anonymous
331 Guest login ok, send your complete e-mail address as password.
Password:
ftp> cd /network/netman/dlpi
250 CWD command successful.
ftp> bin
200 Type set to I.
ftp> get dlpi.ps.Z
200 PORT command successful.
150 Opening BINARY mode data connection for dlpi.ps.Z (479345
bytes).
226 Transfer complete.
1476915 bytes received in 39.12 seconds (11.97 Kbyte/s)
ftp> quit
221 Goodbye.
```

There is no guarantee that public internet servers will always be available. If none of the above public internet server hosts are available, you might try using one of the internet archive server listing services, such as, Archie, to search for a public server that has the DLPI specifications.

Writing or Porting Socket Network Protocols

Socket protocols, including the system provided TCP/IP and XNS protocols, are implemented as loadable kernel extensions in AIX Version 4.1. The following general guidelines apply:

- By convention, protocol kernel extensions are installed in the **/usr/lib/drivers** directory. Some system utilities may assume that this is the case.
- When building the kernel extension the relevant base system exports must be imported. The system provided protocols import the following: **kernex.exp**, **syscalls.exp**, **sockets.exp** and **statcmd.exp**. The system export files are located in the **/usr/lib** directory. These exports should provide all of the services and data structures required to port standard BSD socket-based protocols.
- If other kernel extensions are using services provided by the new protocol then users must create an export file and export these services. See **/usr/lib/netinet.exp** for an example of a protocol export file.
- Protocol writers must decide how much of the kernel extension to pin. The major consideration is that the system interrupt handlers will call the protocol's handler with interrupts disabled. (By the way, this is not true for the protocol's fast and slow timers.) So at minimum, the protocol's interrupt handler must be pinned. The system provided socket protocols are pinned in their entirety.

Initialization

There are two phases to socket protocol initialization in AIX. The first phase involves execution of the kernel extension's configuration entry point. This function is designated when the kernel extension is built and is called by the system when the kernel extension is loaded. This function should perform the following tasks:

- Do any lock initialization required by the protocol in an MP environment.
- Do any pinning of modules or data structures required by the protocol**.**
- Add the protocol's communication domain to the system list using the **domain_add** kernel service. **domain_add** will call the domain's initialization function plus inititalization function of all of the protocol's listed in the domain's protocol switch table. The later step is phase two of protocol initialization. Definitions related to this are in **/usr/include/sys/domain.h** and **/usr/include/sys/protosw.h**
- Register address resolution functions, loopback handlers and address resolution IOCTLs with the systen using the **nd_config_proto** kernel service.

  This is illustrated in sample code for a socket protocol's configuration entry point function, on page 14-17.

  The second phase of protocol initialization occurs when the protocol initialization function is called by **domain_add**. This is exactly analogous to what happens on BSD systems. Generally, these initialization procedures should contain all of the protocol initialization procedures which are not AIX specific. Protocols being ported from Berkeley systems will require no changes to their initialization procedures with two exceptions:

- If protocol interrupts are to be scheduled using the AIX software interrupt facilities, then this should be initialized at this point. This is done via a call to the **netisr_add** kernel service.
- Perform any lock initialization required by the protocol in an MP environment.

  This is illustrated in sample code for a socket protocol's initialization function, on page 14-18.

After this initialization process, the socket protocol is loaded into the kernel, configured into the system's socket framework and initialized. All that remains is for the protocol to notify the system of the types of network packets it wishes to receive, from which network interfaces and the format in which packets are to be exchanged with the system. This is accomplished through a call to the **ns_add_filter** kernel service. Because AIX supports multiple protocols concurrently on the same network adapter and because a protocol may not want to receive packets from all of the network interfaces configured into the system, socket protocols should perform this registration when an address is bound to a network interface.

This is illustrated in sample code for a socket protocol's packet registration function, on page 14-18.

Loading

The system provided socket protocols are loaded and configured by the **ifconfig** command. Because **ifconfig** requires prior knowledge of all the communication address families in the system, it cannot be used to load and configure user written socket protocols. The user must provide a configuration command. The basic logic of this command should be as follows:

1. Check if the protocol is loaded using the **sysconfig** kernel service. If not, load the protocol using the **sysconfig** kernel service.
2. Open a socket**.**
3. Issue the appropriate socket IOCTL. For additional information, see **/usr/include/sys/ioctl.h** and the reference articles for the socket IOCTLs.

Socket – Protocol Interface

The interfaces that a socket protocol must support are described in the protocol switch structure, defined in **/usr/include/sys/protosw.h**. These interfaces, along with their call semantics, are:

```
void pr_input(defined per communications domain),

int pr_output(defined per communications domain),

void pr_ctlinput(int cmd, struct sockaddr *sa, caddr_t arg)
     /* cmd is one of the PRS commands listed in protosw.h,
        sa is a sockaddr, and
        arg is an optional argument used within
           the protocol family
     */


int pr_ctloutput(int req, struct socket *so,int level,
                 int optname,mbuf **optval)
    /* req is a PRCO action listed in protosw.h,
       so is a socket,
       level is an indication of which protocol layer,
       optname is a protocol dependent request value,
       optval is for return results
    */
```

```
int pr_usrreq(struct socket *so, int req, struct mbuf *m,
              struct mbuf *nam, struct mbuf *control)
    /* so is the socket,
       req is a PRU request listed in protosw.h,
       m is an optional message chain,
       nam is an optional mbuf containing an address,
       control is an optional mbuf containing control information
           void pr_init(void),
           void pr_fastimo(void),
           void pr_slowtimo(void),
           void pr_drain(void).
    */
```

The socket system calls interact with the protocol solely through the protocol switch structure.

The **pr_init** function is called by the system when the protocol is loaded. After this is accomplished, the system will call the **pr_fastimo** function on a 200 millisecond timer and the **pr_slowtimo** function on a 500 millisecond timer. Unlike other BSD-based systems, AIX calls the protocol's **pr_drain** functions when the system detects a shortage of network memory buffers (mbufs).

Protocols pass data among themselves (for example ip to tcp) using the **pr_input** and **pr_output** functions. **pr_output** moves data towards the network interface and **pr_input** moves data towards the socket system call interface. Control information is passed using the **pr_ctlinput** and **pr_ctloutput** functions. Unlike with **pr_output**, the socket system routines will pass control data down to the protocols using **pr_ctloutput**. The **getsockopt** and **setsockopt** socket system calls are implemented in this fashion.

With the two exceptions noted above, all of the socket-to-protocol interfaces in the system are implemented using **pr_usrreq**. The specific call semantics for each socket system call are as follows:

This is the semantics of the **socket** system call:

```
(pr_usrreq)((struct socket *) so, PRU_ATTACH,
        (struct mbuf *)0, (struct mbuf *)proto,
         (struct mbuf *)0);
```

This is the semantics of the **bind** system call:

```
(pr_usrreq)((struct socket *) so, PRU_BIND,
            (struct mbuf *)0, nam, (struct mbuf *)0);
```

This is the semantics of the **listen** system call:

```
(pr_usrreq)((struct socket *) so, PRU_LISTEN,
            (struct mbuf *)0, (struct mbuf *)0,
            (struct mbuf *)0);
```

This is the semantics of the **close, disconnect** system call:

```
(pr_usrreq)((struct socket *) so, PRU_DISCONNECT,
            (struct mbuf *)0, (struct mbuf *)0,
            (struct mbuf *)0))
```

This is the semantics of the **close** system call:

```
(pr_usrreq)((struct socket *) so, PRU_DETACH,
            (struct mbuf *)0, (struct mbuf *)0,
            (struct mbuf *)0);
```

This is the semantics of the **soabort** system call:

```
(pr_usrreq)((struct socket *) so, PRU_ABORT,
            (struct mbuf *)0, (struct mbuf *)0,
            (struct mbuf *)0);
```

This is the semantics of the **accept** system call:

```
(pr_usrreq)((struct socket *) so, PRU_ACCEPT,
            (struct mbuf *)0, nam, (struct mbuf *)0);
```

This is the semantics of the **connect** system call:

```
(pr_usrreq)((struct socket *) so, PRU_CONNECT,
            (struct mbuf *)0, nam, (struct mbuf *)0);
```

This is the semantics of the **socketpair** system call:

```
(pr_usrreq)(so1, PRU_CONNECT2,
            (struct mbuf *)0, (struct mbuf *)so2,
            (struct mbuf *)0);
```

The semantics for **send, sendto, sendmsg, write** is one of the following forms:

```
(pr_usrreq)((struct socket *) so, PRU_SENDOOB,top,
             addr, control);
      /* or   */
(pr_usrreq)((struct socket *) so, PRU_SEND,top, addr, control);
```

The semantics for **receive, recvfrom, recvmsg, read** is one of the following forms:

```
(pr->pr_usrreq)((struct socket *) so, PRU_RCVOOB, m,
          (struct mbuf *)(flags & MSG_PEEK), (struct mbuf *)0);
      /* or  */
(pr_usrreq)((struct socket *) so, PRU_RCVD, (struct mbuf *)0,
            (struct mbuf *)flags, (struct mbuf *)0);
```

This is the semantics of the **shutdown** system call:

```
 (pr_usrreq)((struct socket *) so, PRU_SHUTDOWN,
            (struct mbuf *)0, (struct mbuf *)0,
            (struct mbuf *)0);
```

This is the semantics of the **setsockopt** system call:

```
(pr_ctloutput) (PRCO_SETOPT, (struct socket *) so, level,
                optname, &m0);
```

This is the semantics of the **getsockopt** system call:

```
(pr_ctloutput) (PRCO_GETOPT, (struct socket *) so, level,
                optname, mp);
```

This is the semantics of the **ioctl** system call:

```
(pr_usrreq)((struct socket *) so, PRU_CONTROL,
     (struct mbuf *)cmd, (struct mbuf *)data, (struct mbuf *)0));
```

This is the semantics of the **stat** system call:

```
(pr_usrreq)((struct socket *) so, PRU_SENSE,
            (struct mbuf *)ub, (struct mbuf *)0,
            (struct mbuf *)0));
```

This is the semantics of the **getsockname** system call:

```
(pr_usrreq)((struct socket *) so, PRU_SOCKADDR,
            (struct mbuf *)0, m, (struct mbuf *)0);
```

This is the semantics of the **getpeername** system call:

```
(pr_usrreq)((struct socket *) so, PRU_PEERADDR,
            (struct mbuf *)0, m, (struct mbuf *)0);
```

The pr_flags field in the protocol switch table describe basic characteristics of the protocol and impact the behavior of the socket interface. Valid values for these flags are listed in **/usr/include/sys/protosw.h**. If PR_CONNREQUIRED is set then the socket calls will not attempt to transfer data before a connection is established. The PR_ADDR flag causes receive data to be preceded by the senders address. The PR_ATOMIC flag causes sends to be performed in a single protocol send request. The PR_WANTRCVD flag causes the socket routines to notify the protocol when the user has removed data from the socket receive queue. The notification is one of the following socket system calls:

```
(pr->pr_usrreq)((struct socket *) so, PRU_RCVOOB, m,
                (struct mbuf *) (flags & MSG_PEEK),
                (struct mbuf *)0);

(pr_usrreq)((struct socket *) so, PRU_RCVD,
            (struct mbuf *)0, (struct mbuf *)flags,
            (struct mbuf *)0);
```

The PR_RIGHTS flag indicates that the protocol supports the passing of access rights.

The Design and Implementation of the 4.3 BSD UNIX Operating System contains additional information on the socket-to-protocol interface.

Protocol – Socket Interface

The AIX kernel establishes a socket structure (refer to **/usr/include/sys/socketvar.h**) for all sockets. This structure contains send and receive buffer queues (**so_snd** and **so_rcv**), a pointer to the protocol's switch table (**so_proto**) and a pointer to the protocol's control block (**so_pcb**). Protocols establish a back pointer to the socket in the control block (for an example, refer to **/usr/include/netinet/in_pcb.h**) which completes the cross linkage. The later three pointers, **so_proto**, **so_pcb** and the protocol's back pointer are established during the socket system call and the resultant PRU_ATTACH pr_usrreq. It is through these components of the socket structure that the system's protocol-to-socket interface is largely implemented.

On sending of data, reliable protocols typically use the socket send buffer to hold data until acknowledgment. Data is copied from the send buffer using **m_copy** for output. When an acknowledgement is received the protocol removes data from the send buffer with sbdrop or sbdroprecord.

On receipt of data, protocols employ the **sbappend** system services to append data to the appropriate socket's receive buffer. Typically, **sbappend** or **sbappendrecord** are called after the protocol checks that enough space is available in the receive buffer. This check is performed using the **sbspace** kernel service. **sbappendrecord** differs from sbappend in that the data is treated as being the beginning of a new record. Protocols needed to add either access rights or the sender's address to the receive data employ the **sbappendaddr** or **sbappendrights** kernel services. For access rights plus data **sbappendcontrol** should be used. For sender's address, plus access rights (optional), plus data **sbappendaddr** should be employed. Unlike **sbappend** or **sbappendrecord**, these two kernel services check receive buffer space for the caller. These system services do not wake up waiting receivers, so the protocol must issue an **sorwakeup**.

The following sample code illustrates adding data to a socket receive buffer:

```
...
    /*
     * Locate pcb for datagram.
     */
    nsp = ns_pcblookup(&idp->idp_sna, idp->idp_dna.x_port, NS_WILDCARD);
    /*
     * Switch out to protocol's input routine.
     */
    nsintr_swtch++;
...
    idp_input(m, nsp);
...
idp_input(m, nsp)
    struct mbuf *m;
    register struct nspcb *nsp;
{
    register struct idp *idp = mtod(m, struct idp *);
    struct ifnet *ifp = m->m_pkthdr.rcvif;

...
    /*
     * Construct sockaddr format source address.
     * Stuff source address and datagram in user buffer.
     */
...
    if ( ! (nsp->nsp_flags & NSP_RAWIN) ) {
        m->m_len -= sizeof (struct idp);
        m->m_pkthdr.len -= sizeof (struct idp);
        m->m_data += sizeof (struct idp);
    }
    if (sbappendaddr(&nsp->nsp_socket->so_rcv,(struct sockaddr
*)&idp_ns,
        m, (struct mbuf *)0) == 0)
        goto bad;
    sorwakeup(nsp->nsp_socket);
    return;
bad:
    m_freem(m);
}
```

Protocol – Network Interface

In AIX Version 4.1, paths through which socket messages can be sent and received are configured into the system via network interfaces. Normally, a hardware or pseudo device is associated with each interface. An interface and its addresses are defined by kernel **ifnet** structures (refer to **/usr/include/net/if.h**). The linked **ifnet** structures provide a list of **socket** interface names and protocol addresses configured in the system. It is through these **ifnet** structures that the system's protocol-to-network interface is largely implemented. This approach provides the socket protocols with a consistent interface to all network hardware devices.

Interface properties, including state information, are conveyed to the protocols through the ifnet flags and maximum transmission unit (mtu) fields. The most important interface flags and their meanings are:

IFF_UP Interface is up and available for protocol use.

IFF_BROADCAST
Interface is broadcast capable.

IFF_LOOPBACK
Interface is a software loopback.

IFF_POINTTOPOINT
: Interface is a point-to-point link (slip).

IFF_RUNNING
: Interface resources have been allocated.

IFF_NOARP
: Interface should not use address-resolution protocol.

IFF_SIMPLEX
: Interface cannot hear its own transmissions.

IFF_DO_HW_LOOPBACK
: Bypass software loopback.

IFF_ALLCAST
: Token-ring only. Sets all rings broadcast.

IFF_SNAP
: Ethernet only. Interface is 802.3.

Protocols pass data to the network interface employing the **if_output** routine. Each network interface accepts output datagrams of a specified (via the mtu) maximum length. Output occurs when **if_output** is called as follows:

```
(*ifp->if_output)(struct ifnet *ifp, struct mbuf *m, struct
sockaddr *dst, struct rtentry *rt)
```

This has the following parameters:

- `ifp` is the **ifnet** pointer for the interface.
- `m` is the **mbuf** chain to be sent.
- `dst` is the destination address.
- `rt` is an optional routing entry.

The network interface is responsible for encapsulation or decapsulation of any link-layer protocol headers required to deliver the message. This resolution is accomplished as follows:

- If dst–>sa_family is equal to AF_UNSPEC, the link-layer header is copied from dst–>sa_data. In this case, the protocol provided the link-layer header.
- If dst–>sa_family is equal to AF_UNSPEC, then the network interface calls the protocol's address resolution routine. This routine was registered when the protocol was initialized (see the heading "Initialization" in "Writing or Porting Socket Network Protocols", on page 14-8).

The format of the call to the protocol's address resolution routine is:

```
(*dst->af_family.resolve) (struct arpcom *ac, struct mbuf *m,
struct sockaddr *dst, caddr_t *llh)
```

The call has the following parameters:

- `m` is the mbuf chain to be sent,
- `dst` is the destination address
- `llh` is the link layer header to be filled out by the resolution routine. `llh` must be big enough to hold the largest possible link layer header.

The network interface module generally maintains the **ifnet** data structure as part of a larger data structure (an **arpcom**) that contains interface specific information. Thus `ac` is typically set to `(struct arpcom *) ifp`, where `ifp` is the **ifnet** pointer for the interface. Note that AIX Version 4.1 provides several generic address resolution routines which may be employed by protocols.

Protocols pass control data to the network interface employing the **if_ioctl** routine. Most importantly, interface addresses are set with IOCTL requests. The IOCTL requests to set interface addresses, (SIOCSIFADDR, SIOCSIFDSTADDR), and to set and delete multicast addresses (SIOCADDMULTI, SIOCDELMULTI) generally require work at the interface layer and should be passed along by the protocols with an **if_ioct**l. The specific format of this call is:

```
(*ifp->if_ioctl)(struct ifnet *ifp, int cmd, caddr_t data)
```

This call has the following parameters:

- `ifp` is the **ifnet** pointer for the interface.
- `cmd` is the IOCTL.
- `data` is the IOCTL data.

Network – Protocol Interface

In AIX Version 4.1, receive data is passed directly from the common data link interface to the protocols bypassing the network interface layer. The protocol registers with the system the format and the method of delivery for input packets. This registration is done through the **ns_add_filter** kernel service at the protocol's initialization. By selecting the input packet format, the protocol informs the system whether to strip various portions of the link layer header before receive data is presented. There are two methods of delivery:

- A direct call from the interrupt level to the protocol's input function
- An enqueue of the packet on the protocol's input queue and a schedule of the appropriate software interrupt

When called directly, the format of the call to the protocol is:

```
(*protocol_input)(struct ndd *ndd,  struct mbuf *data, caddr_t
*llc, caddr_t *protocookie)
```

The parameters are:

- `ndd` is an **ndd** pointer (see **/usr/include/sys/ndd.h**) to the receiving interface.
- `data` is an **mbuf** pointer to the received data (in the format requested).
- `llc` is a pointer to the link-layer header of the received packet.
- `protocookie` is an address passed in by the protocol when it registered to receive packets of this type. This can be useful for demuxing purposes.

When the packet is enqueued, the protocol receives only the data, in the format requested.

IP Encapsulation/Adding Protocols to the System IP Protocol Switch

The **domain_add** kernel service can be used for adding an entire communications address family with its own protocol switch (see the heading “Initialization” in “Writing or Porting Socket Network Protocols”, on page 14-8). To add entries to an existing protocol switch (for example, IP encapsulation or a new protocol within IP) use the **protosw_enable** kernel service. To remove the protocol switch use the **protosw_disable** kernel service. These services currently only support the AF_INET communications domain. The user is responsible for pinning the new protocol switch entry.

The following sample code illustrates adding IP protocol switch entries to the system:

```
/* set up protocol switch table in internet protocol */
{
    extern int protosw_enable();
    struct protosw *pr;
    struct protosw xns_sw = { SOCK_RAW,0,IPPROTO_IDP,
                          PR_ATOMIC|PR_ADDR,idpip_input,0,
                          nsip_ctlinput,0,0,0,0,0,0,};
    pr = pffindproto(PF_INET, IPPROTO_RAW, SOCK_RAW);
    if ( pr != 0 ) {
        /* enable the protocol switch so that the IP will handle
         * the incoming xns encapsulating packet and pass along.
         * The route pointers are passed because they are not
         * resolve at load time.
         */
        protosw_enable(&xns_sw);
               /* XXX Should check the rc and do ? */
    }
}
```

Sample Socket Protocol

This sample socket protocol includes the following pieces of sample code:

- Configuration entry point function
- Initialization function
- Packet registration function

Sample Socket Protocol's Configuration Entry Point Function

```
struct protosw nssw[] = {
{ 0,&nsdomain,0,0, 0,idp_output,0,0, 0, ns_init,0,0,0,},
{SOCK_DGRAM,&nsdomain,0,PR_ATOMIC|PR_ADDR,0,0,idp_ctlinput,idp_ctloutput,
                  idp_usrreq,0,0,0,0,},
{SOCK_STREAM,&nsdomain,NSPROTO_SPP,PR_CONNREQUIRED|PR_WANTRCVD,spp_input,0,
                 spp_ctlinput,spp_ctloutput, spp_usrreq,
                 spp_init,spp_fasttimo,spp_slowtimo,0,},
{ SOCK_SEQPACKET,&nsdomain,NSPROTO_SPP, PR_CONNREQUIRED|PR_WANTRCVD|PR_ATOMIC,
               spp_input,0,spp_ctlinput,spp_ctloutput, spp_usrreq_sp,0,0,0,0,},
{SOCK_RAW,&nsdomain,NSPROTO_RAW,PR_ATOMIC|PR_ADDR,idp_input,idp_output,0,
                                  idp_ctloutput,  idp_raw_usrreq,0,0,0,0,},
{ SOCK_RAW,&nsdomain,NSPROTO_ERROR,PR_ATOMIC|PR_ADDR,idp_ctlinput,idp_output,0,
                                idp_ctloutput,  idp_raw_usrreq,0,0,0,0,},
}
struct domain nsdomain =
    { AF_NS, "network systems", 0, 0, 0,
      nssw, &nssw[sizeof(nssw)/sizeof(nssw[0])],
      0, 0, ns_funnel, ns_funfrc };
. . .
;/*
 *
 * config_ns - entry point for netns kernel extension
 *
 */
config_ns(cmd, uio)
    int cmd;
    struct uio *uio;
{
    int err, nest;
    struct config_proto config_proto;
    err = 0;
    nest = lockl(&kernel_lock, LOCK_SHORT);
switch (cmd) {
    case CFG_INIT:
        /* check if kernel extension already loaded */
        if (extension_loaded)
            goto out;
        ns_lock_init();
/*
 * pin the netns kernel extension
*/
         if (err = pincode(config_ns))
            goto out;       /* Add ns domain */
        domain_add(&nsdomain);
        config_proto.loop = nsintr;
        config_proto.loopq = &nsintrq;
        config_proto.netisr = NETISR_NS;
        config_proto.resolve = ns_arpresolve;
        config_proto.ioctl = NULL;
        config_proto.whohas = NULL;
        nd_config_proto(AF_NS, &config_proto);
        extension_loaded++;
        break;
```

```
    case CFG_TERM:
    default:
        err = EINVAL;
    }out:
    if (nest != LOCK_NEST)
        unlockl(&kernel_lock);

    return(err);
}
```

Sample Socket Protocol's Initialization Function

```
      void
      ns_init()
      {
              struct timestruc_t ct;
              extern void curtime();
          IFQ_LOCK_DECL() ns_broadhost = * (union ns_host *) allones;
          ns_broadnet = * (union ns_net *) allones;
          nspcb.nsp_next = nspcb.nsp_prev = &nspcb;
          nsrawpcb.nsp_next = nsrawpcb.nsp_prev = &nsrawpcb;
          nsintrq.ifq_maxlen = nsqmaxlen;
          curtime(&ct);   /* get the current system time */
          ns_pexseq = ct.tv_nsec/1000;    /* use microsecond as seq no. */
          ns_netmask.sns_len = 6;
          ns_netmask.sns_addr.x_net = ns_broadnet;
          ns_hostmask.sns_len = 12;
          ns_hostmask.sns_addr.x_net = ns_broadnet;
          ns_hostmask.sns_addr.x_host = ns_broadhost;
          IFQ_LOCKINIT(&nsintrq);
           rtinithead(AF_NS, 16, setnsroutemask);
          (void) netisr_add(NETISR_NS, nsintr, &nsintrq, &nsdomain);
      }
```

Sample Socket Protocol's Packet Registration Function

```
    /*
     * Generic internet control operations (ioctl's).
     */
    ns_control(so, cmd, data, ifp)
        struct socket *so;
        int cmd;
        caddr_t data;
        register struct ifnet *ifp;
    {
        register struct ifreq *ifr = (struct ifreq *)data;
        register struct ns_aliasreq *ifra = (struct ns_aliasreq *)data;
        register struct ns_ifaddr *ia;
        struct ifaddr *ifa;
        struct ns_ifaddr *oia;
        struct mbuf *m;
        int error, dstIsNew, hostIsNew;

        . . .

    switch (cmd) {      .
     . .    case SIOCSIFADDR:
            return (ns_ifinit(ifp, ia, (struct sockaddr_ns *)
                                &ifr->ifr_addr, 1));
            . . .
        }
        . . .
    }
```

```
/*
 * Initialize an interface's internet address
 * and routing table entry.
 */
ns_ifinit(ifp, ia, sns, scrub)
    register struct ifnet *ifp;
    register struct ns_ifaddr *ia;
    register struct sockaddr_ns *sns;
{
    struct sockaddr_ns oldaddr;
    register union ns_host *h = &ia->ia_addr.sns_addr.x_host;
    int error;
        . . .
    return(ns_ns_filter(ifp));
}
ns_ns_filter(ifp)
struct ifnet    *ifp;
{
        struct ns_user  ns_user;
        struct ns_8022  filter;
        struct ndd      *nddp;
    char            ifname[IFNAMSIZ];
    int
    rc;
    /*
     * Alloc the ndd.  Note that we never free it!!
     */
    sprintf(ifname, "%s%d", ifp->if_name, ifp->if_unit);
    if (rc = ns_alloc(ifname, &nddp))
        return(rc);
    /*
     * Add 802.3 filter.
     */
    bzero(&filter, sizeof(filter));
     filter.filtertype = NS_8022_LLC_DSAP;
    filter.dsap = DSAP_XNS;
     ns_user.isr = nsintr;
     ns_user.protoq = &nsintrq;
    ns_user.netisr = NETISR_NS;
    ns_user.pkt_format = NS_PROTO;
    ns_user.ifp = ifp;
    rc = ns_add_filter(nddp, &filter, sizeof(filter), &ns_user);
    return(rc);
}
```

Sample Code for Direct Access to Device Driver via STREAMS

Sample client and server source code demonstrating direct user access to device drivers via STREAMS can be found in directory /**usr**/**samples**/**dlpi**/*.

Chapter 15. Debugging Tools

This chapter provides information about the available procedures for debugging a device driver which is under development. The procedures discussed include:

- Saving device driver information in a system dump, on page 15-1.
- Using the **crash** command to interpret and format system structures, on page 15-5.
- Using the kernel debugger to set breakpoints and display variables and registers, on page 15-25.
- Error logging to record device-specific hardware or software abnormalities, on page 15-78.
- Using the **trace** facility to monitor entry and exit of device drivers and selectable system events, on page 15-86.

System Dump

The system dump copies selected kernel structures to the dump when an unexpected system halt occurs, when the reset button is pressed, or when the special system dump key sequences are entered. You can also initiate a system dump through the System Management Interface Tool (SMIT). For more information, see “Start a System Dump” in *AIX Problem Solving Guide and Reference*.

The dump device can be dynamically configured, which means that either the tape or logical volumes on hard disk can be used to receive the system dump. Use the **sysdumpdev** command to dynamically configure the dump device.

You can also define primary and secondary dump devices. A primary dump device is a dedicated dump device, while a secondary dump device is shared.

The system kernel **dump** routine contains all the vital structures of the running system, such as the process table, the kernel’s global memory segment, and the data and stack segment of each process.

Be sure to refer to the system header files in the /**usr**/**include**/**sys** directory. The name of the file tells which structure and associated information it contains. For example, the user block is defined in **sys**/**user.h**. The process block is defined in **sys**/**proc.h**.

When you examine system data that maps into these structures, you can gain valuable kernel information that can explain why the dump was called.

Initiating a System Dump

A system dump initiated by a kernel panic is written to the primary dump device. If you initiate a system dump by pressing the reset button, the system dump is written to the primary dump device.

Use the special key sequences to determine whether the write of a system dump goes to the primary dump device or to the secondary dump device. To write to the primary dump device, use the sequence Ctrl-Alt-NumPad1. To write to the secondary dump device, use the sequence Ctrl-Alt-NumPad2.

To use SMIT, select **Problem Determination** from the main menu, then select **System Dump**. This presents a menu that allows you to initiate a system dump to either the primary or secondary device, and manipulate the dump devices and the system dump files.

If you prefer to initiate the system dump from the command line, use the **sysdumpstart** command. Use the **–p** flag to write to the primary device or the **–s** flag to write to the secondary device.

If you want your device to be a primary or secondary device, the driver must contain a **dddump** routine. For more information, see the “dddump Entry Point” section in Chapter 4.

When the system dump completes, the system either halts or reboots, depending upon the setting of the **autorestart** attribute of `sys0`. This can be shown and altered using SMIT by selecting **System Environments**, then **Change** / **Show Characteristics of Operating System**. The **Automatically REBOOT system after a crash** item shows and sets this value.

Including Device Driver Information in a System Dump

The system dump is table driven. The two parts of the table are:

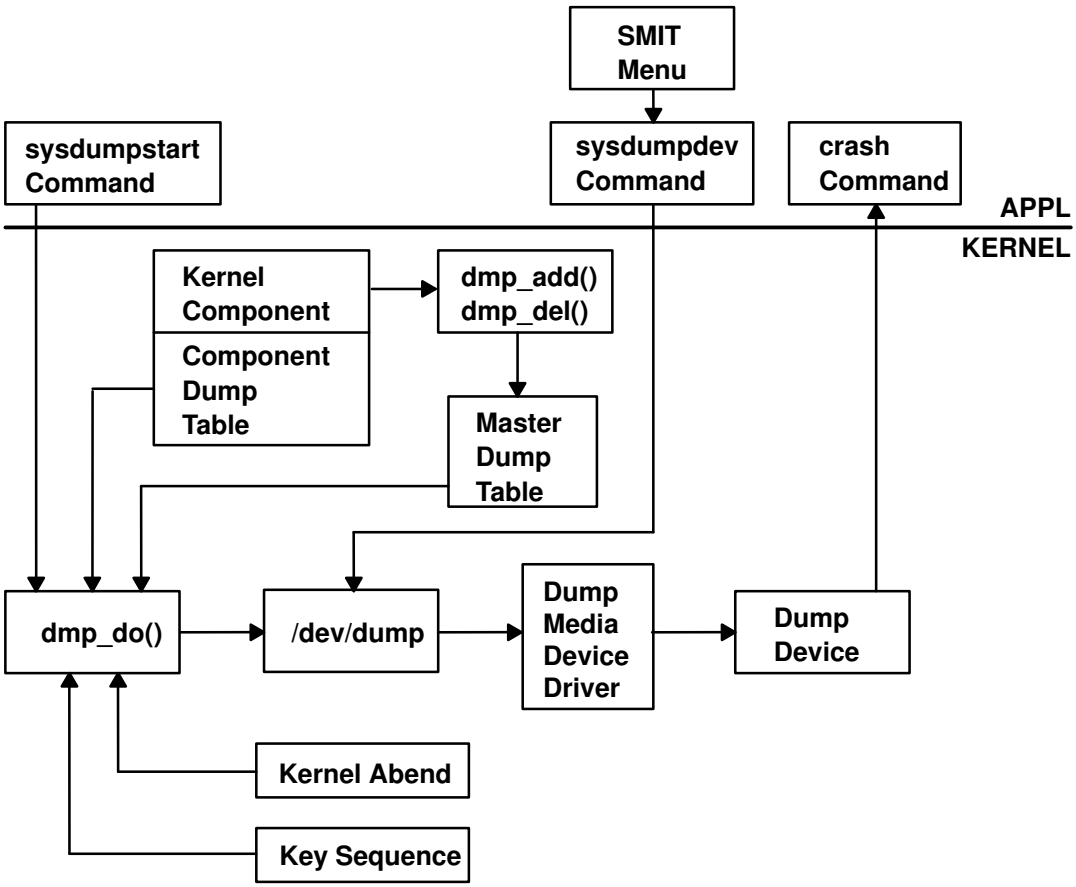
master dump table
: A master dump table entry is a pointer to a function which is provided by the device driver. The function is called by the kernel dump routine when a system dump occurs. The function must return a pointer to a component dump table.

component dump table
: Specifies memory areas to be included in a system dump.

Both the master dump table and the component dump table must reside in pinned global memory.

When a dump occurs, the kernel dump routine calls the function pointed to in the master dump table twice. On the first call, an argument of 1 indicates that the kernel dump routine is starting to dump the data specified by the component dump table.

On the second call, an argument of 2 indicates that the kernel dump routine has finished dumping the data specified by the component dump table. The component dump table should be allocated and pinned during initialization. The entries in the component dump table can be filled in later. The function pointed to in the master dump table must not attempt to allocate memory when it is called. The following System Dump Flow figure shows the flow of a system dump.



System Dump Flow

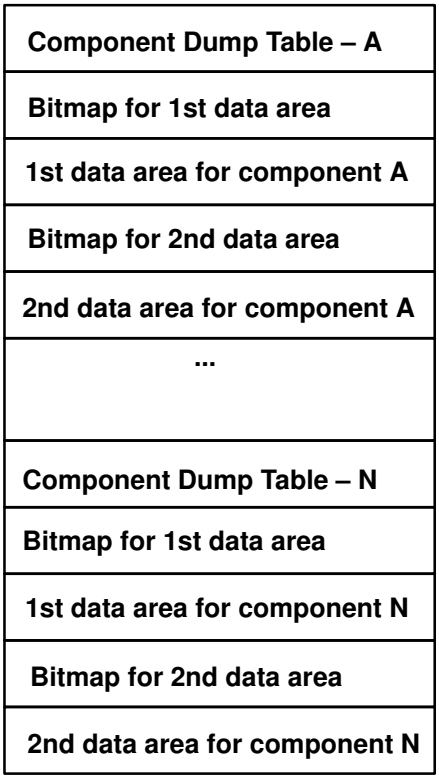
In order to have your device driver data areas included in a system dump, you must register the data areas in the master dump table. Use the **dmp_add** kernel service to add an entry to the master dump table. Conversely, use the **dmp_del** kernel service to delete an entry from the master dump table. The syntax is as follows:

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

int dmp_add(cdt_func) or int dmp_del(cdt_func)
int cdt * ((*cdt_func) ());
```

The **cdt** structure is defined in the **sys/dump.h** header file. A **cdt** structure consists of a fixed-length header (**cdt_head** structure) and an array of one or more **cdt_entry** structures.

The **cdt_head** structure contains a component name field, containing the name of the device driver, and the length of the component dump table. Each **cdt_entry** structure describes a contiguous data area, giving a pointer to the data area, its length, a segment register, and a name for the data area. Use the name supplied for the data area to refer to it when the **crash** command formats the dump. The following Kernel Dump Image figure illustrates a dump image.



Kernel Dump Image

Formatting a System Dump

Each device driver that includes data in a system dump can install a unique formatting routine in the **/usr/lib/ras/dmprtns** directory. A formatting routine is a command that is called by the **crash** command. The name of the formatting routine must match the component name field of the corresponding component dump table.

The **crash** command forks a child process that runs the formatting routines. If a formatting routine is not provided for a component name, the **crash** command runs the **_default_dmp_fmt** default formatting routine, which prints out the data areas in hex.

The **crash** command calls the formatting routine as a command, passing the file descriptor of the open dump image file as a command line argument. The syntax for this argument is **–ffile_descriptor**.

The dump image file includes a copy of each component dump table used to dump memory. Before calling a formatting routine, the **crash** command positions the file pointer for the dump image file to the beginning of the relevant component dump table copy.

The dumped memory is laid out in the dump image file with the component dump table and is followed by a bitmap for the first data area, then the first data area itself. A bitmap for the next data area follows, then the next data area itself, and so on.

The bitmap for a given data area indicates which pages of the data area are actually present in the dump image and which are not. Pages that were not in memory when the dump occurred were not dumped. The least significant bit of the first byte of the bitmap is set to 1 if the first page is present. The next least significant bit indicates the presence or

absence of the second page, and so on. A macro for determining the size of a bitmap is provided in **sys**/**dump.h**.

The crash Command

The **crash** command is a particularly useful tool for device driver development and debugging, which interprets and formats the system structures. The **crash** command is interactive and allows you to examine an operating system image or an active system. An operating system image is held in a system dump file, either as a file or on the dump device. When you run the **crash** command, you can optionally specify a system image file and kernel file, as shown in the syntax below:

```
crash [-a] [-i IncludeFile] [ SystemImageFile [ KernelFile ] ]
```

The default *SystemImageFile* is **/dev/mem** and the default *KernelFile* is **/usr/lib/boot/unix**.

To run the **crash** command on the active system, enter:

```
crash
```

Because the command uses **/dev/mem**, you need root permissions.

To invoke the **crash** command on a system image file, enter:

```
crash SystemImageFile
```

where *SystemImageFile* is either a file name or the name of the dump device.

Note that by convention the symbol names for function entry points always begin with a . (period), while symbol names for data areas always begin with an _ (underscore). There is usually a data address corresponding to an external entry point address, and the **od** subcommand displays the data address for a name with no prefix. To be safe, use the proper prefix when looking for addresses.

Use the **–a** flag to generate a list of data structures without using subcommands. The resulting list is large, so you can redirect the output to either a file or to a printer.

Use the **–i** flag to read the given include file, allowing the **print** subcommand to output data according to the include file structures.

You can use a variety of subcommands to view the system structures. These subcommands can have flags that modify the format of the data. If you do not use a flag to specify what you want to see, all valid entries are displayed.

crash Subcommands

Once you initiate the **crash** command, **>** is the prompt character. For a list of the available subcommands, type the **?** character. To exit, type **q**. You can run any shell command from within the crash command by preceding it with an ! (exclamation mark).

Since the **crash** command only deals with kernel threads, the word thread when used alone will be used to mean kernel thread in the **crash** documentation that follows. The default thread for several subcommands is the current thread (the thread currently running). On a multiprocessor system, you can use the **cpu** subcommand to change the current processor: the default thread becomes the running thread on the selected processor.

The parameters *ProcessTableEntry* and *ThreadTableEntry* are used in many subcommands to indicate a process or thread respectively. These parameters are simply numbers for table entry indexes which can be displayed using the **proc** and **thread** subcommands.

Note that many structures displayed are longer than one screen length. Make sure that you can halt scrolling if it is important to view something in detail. To do this, use the **stty** command:

```
stty ixon ixany
```

Use the **Ctrl-S** key sequence to stop scrolling and **Ctrl-Q** to resume scrolling.

buf [*BufferHeaderNumber*]

The **buf** subcommand displays the system buffer headers. A buffer header contains the information required to perform block I/O. If you type the **buf** subcommand with no *BufferHeaderNumber*, a summary of the system buffer headers is displayed.

Aliases = **bufhdr**, **hdr**

```
> buf
BUF MAJ  MIN    BLOCK   FLAGS
  0 000a 000b       8   done stale
  1 000a 000b     243   done stale
  2 000a 000b      24   done stale
...
```

If you type the **buf** subcommand with a *BufferHeaderNumber* a single complete header is displayed:

```
> buf 3
BUFFER HEADER 3:
 b_forw:  0x014d0528, b_back:   0x014d0160, b_vp:      0x00000000
 av_forw: 0x014d0160, av_back:  0x014d0528, b_iodone:  0x000185f8
 b_dev:   0x000a000b, b_blkno:           0, b_addr:    0x014e9000
 b_bcount:      4096, b_error:           0, b_resid:           0
 b_work:  0x80000000, b_options:0x00000000, b_event:   0xffffffff
 b_start.tv_sec:             0, b_start.tv_nsec:             0
 b_xmemd.aspace_id: 0x00000000, b_xmemd.subspace_id: 0x00000000
 b_flags: read done stale
```

Refer to the **sys/buf.h** header file for the structure definition.

buffer [*Format*] [*BufferHeaderNumber*]

The **buffer** subcommand displays the data in a system buffer according to the *Format* parameter. When specifying a buffer header number, the buffer associated with that buffer header is displayed. If you do not provide a *Format* parameter, the previous *Format* is used. Valid options are **decimal**, **octal**, **hex**, **character**, **byte**, **i–node**, **directory**, and **write**. The **write** option creates a file in the current directory containing the buffer data.

Aliases = **b**

```
> buffer hex 3

BUFFER FOR BUF_HDR 3
00000: 41495820  4c564342  00006a66  73000000
00020: 00000000  00000000  00000000  00000000
00040: 00000000  00000000  00003030  30303033
...
```

callout

The **callout** subcommand displays all active entries on the active **trblist**. When the **time-out** kernel extension is used in a device driver, this timer request is entered on a system-wide list of active timer requests. This list of timer requests is the **trblist**. Any timer which is active is on this list until it expires.

Aliases = **c**, **call**, **calls**, **time**, **timeout**, **tout**

```
>callout

TRB's On The Active List Of Processor 0.

TRB #1 on Active List
 Timer address.....................0x0
 trb.to_next.......................0x0
 trb.knext.........................0x59aa100
 trb.kprev.........................0x0
 Thread id (-1 for dev drv).........0xfffffffe
 Timer flags.......................0x12
 trb.timerid.......................0x0
 trb.eventlist.....................0xffffffff
 trb.timeout.it_interval.tv_nsec....0x0
 trb.timeout.it_interval.tv_sec.....0x0
 Next scheduled timeout (secs)......0x2d63f6a8
 Next scheduled timeout (nanosecs)..0xc849a80
 Timeout function..................0x8c748
 Timeout function data.............0x59aa040
 TRB #2 on Active List
...
```

Refer to **sys/timer.h** for the structure definitions, and to InfoExplorer for a description of the time-out mechanism.

cm [*ThreadTableEntry SegmentNumber*]

The **cm** subcommand is used by the **od** subcommand to change the current segment map. The **cm** subcommand changes the map of the **crash** command internal pointers for any thread segment not paged out, if you specify the thread *ThreadTableEntry* and *SegmentNumber*. This allows the **od** subcommand to display data relative to the beginning of the segment desired. The following example sets the map to thread *ThreadTableEntry* 3 to *SegmentNumber* 2, then displays ten words starting from the offset 0:

Aliases = none

```
> cm 3 2
t3,2 >> od 0 10
00000000: 00000000 00000000 00000000 00000000
00000010: 00000000 00000000 00000000 00000000
00000020: 00000000 00000000
t3,2 >>
...
```

Using the **cm** subcommand without any parameters resets the map of internal pointers.

cpu [*ProcessorNumber*]

If no argument is given, the **cpu** subcommand displays the number of the currently selected processor. Initially, the selected processor is processor 0. If the *ProcessorNumber* argument is given, the **cpu** subcommand selects the specified processor as the current processor. By extension, this selects the current kernel thread (the running kernel thread on the selected processor). Processor numbering starts from zero.

Aliases = none

```
> cpu
Selected cpu number :  0
```

dblock [*Address*]

The **dblock** subcommand displays the allocated streams data block headers. The address parameter is required. If the address is not supplied, this command will print an error message stating that the address is required. Refer to the **sys/stream.h** file for the `datab` structure definitions. The **freep** and **db_size** definitions are not included in **/usr/include/sys/stream.h**. These structure members are described here:

freep Address of the free pointer

db_size Size of the data block

There is no checking performed on the address passed in as the required parameter. The **dblock** subcommand will accept any address. It is up to the user to be sure that a valid address is specified.

To determine a valid address run the **mblock** subcommand. From the output of the **mblock** subcommand, select a non-zero data block address under the DATABLOCK column heading.

This subcommand can be issued from crash on either a running system or a system dump.

Aliases = dblk

```
> queue 59d5a74
   QUEUE    QINFO     NEXT  PRIVATE  FLAGS     HEAD   OTHERQ      COUNT
 59d5a74  1884c1c  59d5474  59d5500 0x003e  59e1c00  59d5a00       4096
> mblk 59e1c00
 ADDRESS     NEXT PREVIOUS     CONT     RPTR     WPTR DATABLOCK
 59e1c00        0        0        0  59e2000  59e3000   59e1c44
> dblk 59e1c44
 ADDRESS    FREEP     BASE      LIM   REFCNT     TYPE     SIZE
 59e1c44        0  59e2000  59e3000        1        0     1000
```

dlock [*ThreadIdentifier* | **–p** [*ProcessorNumber*]]

Displays deadlock analysis information about all types of locks (simple, complex, and lockl). The **dlock** subcommand searches for deadlocks from a given start point. If *ThreadIdentifier* is given, the corresponding kernel thread is the start point. If **–p** is given without a *ProcessorNumber*, the start point is the running kernel thread on the current processor. If **–p** *ProcessorNumber* is given, the running kernel thread on the specified processor is the start point. If no arguments are given, **dlock** searches for deadlocks among all threads on all processors.

The first output line gives information about the starting kernel thread, including the lock which is blocking the kernel thread, and a stack trace showing the function calls which led to the blocking lock request. Each subsequent line shows the lock held by the blocked kernel thread from the previous line, and identifies the kernel thread or interrupt handler which is blocked by those locks. If the information required for a full analysis is not available (paged out), an abbreviated display is shown; in this case, examine the stack trace to locate the locking operations which are causing the deadlock. The display stops when a lock is encountered for a second time, or no blocking lock is found for the current kernel thread.

Aliases = none

```
>dlock 00d3f
Deadlock from tid 00d3f. This tid waits for the first line lock,
owned by Owner-Id that waits for the next line lock, and so on...
     LOCK NAME    |   ADDRESS  |  OWNER-ID | WAITING FUNCTION
           lockC1 | 0x001f79e0 | Tid  113d | .lock_write_ppc
                    called from : .times + 0000020c
Dump data incomplete.Only 0 bytes found out of 4.
                    called from : .file + 0000000b
           lockC2 | 0x001f79e8 | Tid   d3f | .lock_write_ppc
                    called from : .times + 000001c8
Dump data incomplete.Only 0 bytes found out of 4.
                    called from : .file + 0000000b
```

dmodsw

The **dmodsw** subcommand displays the streams drivers switch table. The information printed is contained in an internal structure. The following members of this internal structure are described here:

address	Address of **dmodsw**
d_next	Pointer to the next driver in the list
d_prev	Pointer to the previous driver in the list
d_name	Name of the driver
d_flags	Flags specified at configuration time
d_sqh	Pointer to synch queue for driver-level synchronization
d_str	Pointer to streamtab associated with the driver
d_sq_level	Synchronization level specified at configuration time
d_refcnt	Number of open or pushed count
d_major	Major number of a driver

The flags structure member, if set, is based one of the following values:

#define	**Value**	**Description**
F_MODSW_OLD_OPEN	0x1	Supports old-style (V.3) open/close parameters
F_MODSW_QSAFETY	0x2	Module requires safe timeout/bufcall callbacks
F_MODSW_MPSAFE	0x4	Non-MP-Safe drivers need funneling

The synchronization level codes are described in the **/usr/include/sys/strconf.h** file.

This subcommand can be issued from crash on either a running system or a system dump.

Aliases = none

```
> dmodsw
NAME        ADDRESS     NEXT PREVIOUS FLAG   SYNCHQ STREAMTAB S-LVL COUNT MAJOR
sad         5a0cf40  5a0cf00  5a0c9c0  0x0  5a0ad40   188c600     3     0    12
slog        5a0cf00  5a0cec0  5a0cf40  0x0  5a0ad20   188c8a0     3     0    13
en          5a0cec0  5a0ce80  5a0cf00  0x0  5a0ad00   1893ee0     3     0    27
et          5a0ce80  5a0ce40  5a0cec0  0x0  5a0ace0   1893ee0     3     0    28
tr          5a0ce40  5a0ce00  5a0ce80  0x0  5a0acc0   1893ee0     3     0    29
fi          5a0ce00  5a0cdc0  5a0ce40  0x0  5a0aca0   1893ee0     3     0    30
echo        5a0cdc0  5a0cd80  5a0ce00  0x0        0   18951a0     5     0    31
nuls        5a0cd80  5a0cd40  5a0cdc0  0x0        0   1895190     5     0    32
spx         5a0cd40  5a0cd00  5a0cd80  0x0  5a0ac80   1895d70     3     0    33
unixdg      5a0cd00  5a0ccc0  5a0cd40  0x0  5a0ac60   18a14e0     3     0    34
unixst      5a0ccc0  5a0cc80  5a0cd00  0x0  5a0ac40   18a14e0     3     0    35
udp         5a0cc80  5a0cc40  5a0ccc0  0x0  5a0ac20   18a14e0     3     0    36
tcp         5a0cc40  5a0cb40  5a0cc80  0x0  5a0ac00   18a14e0     3     0    37
rs          5a0cb40  5a0cb00  5a0cc40  0x0        0   18b31d0     5     1    15
pts         5a0cb00  5a0ca40  5a0cb40  0x0        0   18fc930     4     7    24
ptc         5a0ca40  5a0ca00  5a0cb00  0x0        0   18fa5c0     4     2    23
ttyp        5a0ca00  5a0c9c0  5a0ca40  0x0        0   18fc950     4     0    26
ptyp        5a0c9c0  5a0cf40  5a0ca00  0x0        0   18fc940     4     0    25
```

ds [*Address*]

The **ds** subcommand returns the symbols closest to the given address. The **ds** subcommand can take either a text address or a data address.

Aliases = none

```
> ds 012345
       .ioctl_systrace  + 0x000001b5
```

When a number such as *0x000001b5* is displayed, it shows the number of assembly language instructions by which the given address is offset from the entry point of the routine.

du [*ThreadTableEntry*]

Displays a combined hex and ASCII dump of the specified thread's uthread structure and of the user structure of the process which owns the thread. If the data is not available (paged out), a message is displayed. The default is the current thread.

Aliases = none

```
> du 3
Uthread structure of thread slot  3
  00000000 00000000 00000000 2ff7fec0 00000000 *........./.......*
  00000010 00000303 00000000 00030644 000010b0 *...........D....*
  00000020 22222828 00030644 00006244 00000009 *""((...D..bD....*
.
.
.
```

dump

The **dump** subcommand displays the name of each component for which there is data present. After you select a component name from the list, the **crash** program loads and runs the associated formatting routine contained in the /**usr**/**lib**/**ras**/**dmprtns** directory.

If there is more than one data area for the selected component, the formatting routine displays a list of the data areas and allows you to select one. The **crash** command then displays the selected data area. You can enter the **quit** subcommand to return to the previously displayed list and make another selection or enter **quit** a second time to leave the **dump** subcommand loop.

Aliases = none

Displays messages in the error log. *Count* is the number of messages to print that have already been read by the **errdemon** process. (The default is 3 messages.) **errpt** always prints all messages that have not yet been read by the **errdemon** process.

Aliases = none

file [*FileTableEntry*]

The **file** subcommand displays the file table. Unless you request specific file entries, the command displays only those with a nonzero reference.

Aliases = **files**, **f**

```
> f 3
SLOT  REF     INODE    FLAGS
   3    1  0x018e53f0   read
```

Refer to **sys**/**file.h** for the structure definition.

fmodsw

The **fmodsw** subcommand displays the streams modules switch table. The information printed is contained in an internal structure. The following members of this internal structure are described here:

address	Address of **fmodsw**
d_next	Pointer to the next module in the list
d_prev	Pointer to the previous module in the list
d_name	Name of the module
d_flags	Flags specified at configuration time
d_sqh	Pointer to synch queue for module-level synchronization
d_str	Pointer to streamtab associated with the module
d_sq_level	Synchronization level specified at configuration time
d_refcnt	Number of open or pushed count
d_major	–1

The flags structure member, if set, is based one of the following values:

#define	Value	Description
F_MODSW_OLD_OPEN	0x1	Supports old-style (V.3) open/close parameters
F_MODSW_QSAFETY	0x2	Module requires safe timeout/bufcall callbacks
F_MODSW_MPSAFE	0x4	Non-MP-Safe drivers need funneling

The synchronization level codes are described in the **/usr/include/sys/strconf.h** file.

This subcommand can be issued from crash on either a running system or a system dump.

Aliases = **none**

```
> fmodsw
NAME        ADDRESS     NEXT PREVIOUS FLAG   SYNCHQ STREAMTAB S-LVL COUNT MAJOR
bufcall     5a0cf80  5a0cc00  5a0ca80  0x1  5a0ad60   188bf80     3     0    -1
sc          5a0cc00  5a0cbc0  5a0cf80  0x0  5a0abe0   18a29b0     3     0    -1
timod       5a0cbc0  5a0cb80  5a0cc00  0x0  5a0abc0   18a34b0     3     0    -1
tirdwr      5a0cb80  5a0cac0  5a0cbc0  0x0  5a0aba0   18a4010     3     0    -1
ldterm      5a0cac0  5a0ca80  5a0cb80  0x0        0   18ef460     4     8    -1
tioc        5a0ca80  5a0cf80  5a0cac0  0x0        0   18f0e90     4    10    -1
```

fs [*ThreadTableEntry*]

Traces a kernel stack for the thread specified by *ThreadTableEntry*. Displays the called subroutines with a hex dump of the stack frame for the subroutine that contains the parameters passed to the subroutine. By default, the current thread is traced. This subcommand will not work on a running system because it uses stack tracing; however, it does work on a dump image.

Aliases = none

```
> fs
STACK TRACE:
        **** .et_wait ****
2ff97e78  2FF97ED8 0080D568 00000000 018F4C60   /.^....h......L`
2ff97e88  2FF97EE8 0080D568 00082BC0 000BA020   /.^....h..+.....
2ff97e98  2FF97ED8 28008044 00082418 2FF98000   /.^.(..D..B./...
2ff97ea8  00000000 000B8468 00000000 00000000   .......h........
2ff97eb8  2FF97F38 0000000B 00000004 00000004   /..8............
2ff97ec8  00000005 01DFE258 00000000 E3000600   .......X........
```

inode [–] [<*Major*> <*Minor*> <*INumber*>]

The **inode** subcommand displays the i-node table and the i-node data block addresses. You can display a specific i-node by specifying the major and minor device numbers of the device where the i-node resides and the i-node number. The command displays the i-node only if it is currently on the system hash list.

Aliases = **ino**, **i**

```
>inode
 ADDRESS   MAJ MIN  INUMB REF LINK  UID  GID   SIZE    MODE  SMAJ SMIN FL
AGS
0x018e4e50 010 0007 11264   0    1    2    2     30 ----777    -    -
0x018f9fd0 010 0009 16384   1    6  201    0    512 d---755    -    -
     addr: 16448   0   0   0   0   0   0   0
0x018ea940 010 0011     0   1    0    0    0      0 ----  0    -    -
...
```

kfp [*FramePointer*]

If you use the **kfp** subcommand without parameters, it displays the last kernel frame pointer address that was set using **kfp**. If you specify a frame pointer address, it sets the kernel frame pointer to the new address. Use this subcommand in conjunction with the **–r** flag of the **trace** subcommand.

Aliases = **fp**, **rl**

```
> kfp
```

knlist [*Symbol*]

The **knlist** subcommand displays the addresses of all the specified symbol names. If there is no such symbol, the subcommand displays a `no match` message. Run this subcommand only on an active system.

The **knlist** subcommand runs a subroutine to the active kernel to obtain the address from the system's knlist. The **nm** subcommand provides the same function but searches the symbol table in the Kernel Image File for the address and therefore can be used on a dump.

Aliases = **none**

```
> knlist open
        open:0x000bbc98
```

le [*Module Address*]

The **le** subcommand displays the kernel load list entries. If you specify an address in a kernel extension, the corresponding load list entry is displayed. If you attempt to display a paged out loader entry, the subcommand displays an error message.

Aliases = **none**

linkblk

The **linkblk** subcommand displays the streams linkblk table. Refer to the **/usr/include/sys/stream.h** file for the `linkblk` structure definitions. If there are no `linkblk` structures found on the system, the **linkblk** subcommand will print a message stating that no structures are found.

This subcommand can be issued from crash on either a running system or a system dump.

Aliases = **lblk**

This example shows a regular link:

```
> linkblk
    QTOP     QBOT    INDEX
 5ab8b74  5ae5074  5ab4200
```

This example shows a persistent link:

```
> linkblk
    QTOP     QBOT    INDEX
       0  5ae5174  5a4ef00
```

mblock *Address*

The **mblock** subcommand displays the allocated streams message block headers. The address parameter is required. If the address is not supplied, this command will print an error message stating that the address is required. Refer to the **/usr/include/sys/stream.h** file for the queue structure definitions.

The **mblock** subcommand's checking of the address parameter is limited to verifying that the address falls on a 128-byte boundary. It is up to the user to be sure that a valid address is specified.

To determine a valid address, run the queue subcommand. From the output of the queue subcommand, select a non-zero address for the head of the message queue under the HEAD column heading for either a read queue or a write queue.

This subcommand can be issued from crash on either a running system or a system dump.

Aliases = **mblk**

```
> queue 59d5a74
   QUEUE    QINFO     NEXT  PRIVATE  FLAGS     HEAD   OTHERQ      COUNT
59d5a74  1884c1c  59d5474  59d5500 0x003e  59e1c00  59d5a00       4096
> mblk 59e1c00
 ADDRESS     NEXT PREVIOUS     CONT     RPTR     WPTR DATABLOCK
 59e1c00        0        0        0  59e2000  59e3000   59e1c44
```

mbuf [–] [*Clusters* | *Address ...*]

The **mbuf** subcommand displays the system **mbuf** structures. These structures are memory buffers that are chained and can be manipulated by the Memory Buffer kernel services. If you specify the **–** flag, the subcommand also displays the data associated with the **mbuf** structures.

The **mbuf** subcommand with no additional arguments displays the chain of **mbuf** structures pointed to by the **mbuf** pointer. If you specify *Clusters*, the subcommand displays the chain of **mbuf** structures pointed to by the kernel mbclusters pointer. If you specify *Address*, then the **mbuf** structure at the given address is displayed. Note that valid **mbuf** pointers must be on a 128-byte boundary.

Aliases = **mbuff**

```
> mbuf
  ADDRESS   SIZE    TYPE     LINK       DATAPTR
0x01a67000     0    free  0x00000000  0x01a67000
    DATA:  0x00000000 0x00000000 0x00000000 0x00000000
```

Refer to the **sys**/**mbuf.h** header file for the structure definition.

mst [*Address*]

The **mst** subcommand displays the mstsave portion of the uthread structures at the addresses specified. If you do not specify an address, the subcommand displays information about the last running kernel thread.

Aliases = **none**

ndb

Displays network kernel data structures either for a running system or a system dump. The **ndb** subcommand, short for network debugger, displays the following options:

?	Provides first-level help information.
help	Provides additional help information.
tcb [*Addr*]	Shows TCBs. The default is HEAD TCB.
udb [*Addr*]	Shows UDBs. The default is HEAD UDB.
sockets	Shows sockets from the file table.
mbuf [*Addr*]	Shows the **mbuf** at the specified address.
ifnet [*Addr*]	Shows the **ifnet** structures at the specified address.
quit	Stops the running option.
xit	Exits the **ndb** submenu.

Aliases = none

nm [*Symbol*]

The **nm** subcommand displays the symbol value and type found in *KernelFile*.

Aliases = **none**

```
> nm open
        00095484 000C70  PR SD    <.open>
        00095484         PR LD    .open
        000BBC98 00000C  SV SD    open
```

od [*SymbolName* | *Address*] [*Count*] [*Format*]

The **od** subcommand dumps *Count* number of data values starting at *Symbol* value or *Address* according to *Format*. Possible formats are **octal**, **longoct**, **decimal**, **longdec**, **character**, **hex**, **instruction**, and **byte**. The default is **hex**. Note that if you use the *Format* parameter, you must also use *Count.*

The **od** subcommand is especially useful during program development in order to see structure values at a given point in time.

Aliases = **none**

```
> od open 10
00095484: 7c0802a6 bf21ffe4 90010008 9421ff30
00095494: 609c0000 832202e0 607b0000 60bd0000
000954a4: 63230000 38800000

> od open 10 byte
00095484: 0174 0010 0002 0246 0277 0041 0377 0344
0009548c: 0220 0001

> od 12345
warning: word alignment performed
00012344: 480001d8
```

ppd [*ProcessorNumber* | ***]**

Displays per-processor data area (PPDA) structures for the specified processor. If no processor is specified, the current processor selected by the **cpu** subcommand is used. If the asterisk argument is given, the PPDA of every enabled processor is displayed.

Aliases = none

```
> ppd
Per Processor Data Area for processor  0
csa.....................2fedf500
mstack..................00315db0
fpowner.................e6001360
curthread...............e6001360
r0......................60000000
r1......................6000068e
r2......................d00089b8
r15.....................0000f930
sr0.....................d0005a54
sr2.....................2feac6e0
iar.....................f013c7c4
```

print *type Address*

Does **dbx**-style printing of structures. The **–i** option must be given on the command line to use this feature.

Aliases = none

proc [**–**] [**–r**] [*ProcessTableEntry*]

The **proc** subcommand displays the process table, including the kernel thread count (the number of threads in the process) and state of each process. Use the **–r** flag to display only runnable processes. Use the **–** flag to display a longer listing of the process table.

Aliases = **ps**, **p**

```
>p

SLT ST    PID   PPID   PGRP   UID  EUID  TCNT  NAME
  0 a       0      0      0     0     0     1  swapper
        FLAGS: swapped_in no_swap fixed_pri kproc
  1 a       1      0      0     0     0     1  init
        FLAGS: swapped_in no_swap
  2 a     204      0      0     0     0     1  wait
        FLAGS: swapped_in no_swap fixed_pri kproc
 ...


>p 20

SLT ST    PID   PPID   PGRP   UID  EUID  TCNT  NAME
 20 a    1406      1   1406     0     0     1  ksh
        FLAGS: swapped_in no_swap
```

```
>p -

SLT ST    PID   PPID   PGRP   UID  EUID  TCNT  NAME
  0 a       0      0      0     0     0     1  swapper
        FLAGS: swapped_in no_swap fixed_pri kproc

Links:  *child:0xe3000100  *siblings:0x00000000  *uidl:0xe3001400
    *ganchor:0x00000000
Dispatch Fields:  pevent:0x00000020  wevent:0x00000000
    *p_synch:0xffffffff
Thread Fields:  *threadlist:0xe6000000  threadcount:    1
   active:    1  suspended:    0  local:    0  localsleep:    0
   *synch:0xffffffff
Scheduler Fields:   fixed pri: 16  repage:0x00000000
scount:0x00000000
Misc:  adspace:0x00000808  *ttyl:0x00000000
       *p_ipc:0x00000000  *p_dblist:0x00000000
*p_dbnext:0x00000000
    *lock:0x00000000  kstackseg:0x007fffff *pgrpl:0x08x

Signal Information:
    pending:hi 0x00000000,lo 0x00000000
    sigcatch:hi 0x00000000,lo 0x00000000  sigignore:hi
0xffffffff,lo 0xfff7ffff
Statistics:  size:0x00000000(pages)  audit:0x00000000


SLT ST    PID   PPID   PGRP   UID  EUID  TCNT  NAME
  1 a       1      0      0     0     0     1  init
        FLAGS: swapped_in no_swap

 Links:  *child:0xe3001400  *siblings:0x00000000
*uidl:0xe3000100
    *ganchor:0x00000000
Dispatch Fields:  pevent:0x00000020  wevent:0x00000000
    *p_synch:0xffffffff
Thread Fields:  *threadlist:0xe60000a0  threadcount:    1
   active:    1  suspended:    0  local:    0  localsleep:    0
   *synch:0xffffffff
Scheduler Fields:   nice: 20  repage:0x00000000
scount:0x00000000
Misc:  adspace:0x00000505  *ttyl:0x00000000
       *p_ipc:0x00000000  *p_dblist:0x00000000
*p_dbnext:0x00000000
    *lock:0x00000000  kstackseg:0x007fffff *pgrpl:0x08x
Signal Information:
    pending:hi 0x00000000,lo 0x00000000
    sigcatch:hi 0x00000001,lo 0x18783eff  sigignore:hi
0xfffffffe,lo 0xe787c100
Statistics:  size:0x00000028(pages)  audit:0x00000000
...
```

Refer to the **sys/proc.h** header file for the structure definition.

queue [*Address*]

The **queue** subcommand displays the STREAMS queue. If the address optional parameter is not supplied, crash will display information for all queues available. Refer to the **/usr/include/sys/stream.h** file for the `queue` structure definitions.

If you wish to see the information stored for a read queue, issue the **queue** subcommand with the read queue address specified as the parameter.

When you issue the **queue** subcommand with the address parameter, the column headings do not distinguish between the read queue and the write queue. One queue address will be displayed under the column heading QUEUE. The other queue in the pair will be displayed under the column heading OTHERQ. The write queue will have a numerically higher address than the read queue.

This subcommand can be issued from crash on either a running system or a system dump.

Aliases = **que**

```
> queue
 WRITEQ    QINFO     NEXT  PRIVATE  FLAGS     HEAD    READQ      COUNT
59c2a74  188c50c  59c2474  59b1900 0x002a        0  59c2a00          0
59c2474  18f0e50  59c2274  59b6880 0x0028        0  59c2400          0
59c2274  18ef3d8  59c2174  59cc800 0x0028        0  59c2200          0
59c2174  18b31b4        0  54684f8 0x0028        0  59c2100          0
59d5a74  188c50c  5a94874  59d3c00 0x002a        0  59d5a00          0
5a94874  18f0e50  5a9c074  59ec500 0x0028        0  5a94800          0
5a9c074  18fa748        0  59e5b00 0x0028        0  5a9c000          0
59ff074  188c50c  59eab74  59ffe00 0x002a        0  59ff000          0
59eab74  18f0e50  59ff374  59da500 0x0028        0  59eab00          0
59ff374  18fa748        0  59da380 0x0028        0  59ff300          0
5ab4374  188c50c  59ee174  5ab4c00 0x002a        0  5ab4300          0
59ee174  18f0e50  5ab4774  59da100 0x0028        0  59ee100          0
5ab4774  18ef3d8  5ad2874  59d7800 0x0028        0  5ab4700          0

> queue 5ab4700
  QUEUE    QINFO     NEXT  PRIVATE  FLAGS     HEAD   OTHERQ      COUNT
5ab4700  18ef3bc  59ee100  59d7800 0x0029        0  5ab4774          0
```

quit

Exit from the **crash** command.

Aliases = **q**

qrun

The **qrun** subcommand displays the list of scheduled streams queues. If there are no queues found for scheduling, the **qrun** subcommand will print a message stating there are no queues scheduled for service.

This subcommand can be issued from crash on either a running system or a system dump.

Aliases = **none**

```
> qrun
   QUEUE
 59d5a74
```

socket [–]

The **socket** subcommand displays the system socket structures. Use the **–** flag to also display the socket buffers.

Aliases = **sock**

```
> sock
SLOT:   26  type:0x0002  opts:0x0000  linger:0x0000
     state:0x0080  pcb:0x01d32d8c  proto:0x01c65cf0
     q0:0x00000000  q0len:    0 q:0x00000000
     qlen:    0  qlimit:    0  head:0x00000000
     timeo:    0  error:    0  oobmark:    0  pgrp:    0
...
```

Refer to **sys/socket.h** for structure definitions.

stack [*ThreadTableEntry*]

The **stack** subcommand displays a dump of the kernel stack of the kernel thread identified by *ThreadTableEntry*. The addresses are virtual data addresses rather than true physical addresses. If you do not specify an entry, the subcommand displays information about the last running kernel thread. You cannot trace the stack of the current running kernel thread on a running system.

Aliases = **s**, **stk**, **k**, **kernel**

```
> s 31
KERNEL STACK:
2ff97a50:            8eaa4        16  2ff97ac8         2
2ff97a60:             90b0     8e8b4  2ff97ad8         0
2ff97a70:                1        26  2ff97ac8  2ff98938
...
```

stat

The **stat** subcommand displays statistics found in the dump. These statistics include the panic message (if there is one), time of crash, and system name.

Aliases = **none**

```
> stat
        sysname: AIX
        nodename: funk
        release: 1
        version: 3
        machine: 000003961000
        time of crash: Fri Sep 28 17:50:38 1990
        age of system: 15 day, 6 hr., 25 min.
```

status [*ProcessorNumber*]

Displays a description of the kernel thread scheduled on the designated processor. If no processor is specified, the **status** subcommand displays information for all processors. The information displayed includes the processor number, kernel thread identifier, kernel thread table slot, process identifier, process table slot, and process name.

Aliases = none

```
> status 0
CPU     TID  TSLOT     PID  PSLOT  PROC_NAME
  0    1fe1     31    1fd8     31  crash
```

stream

The **stream** subcommand displays the stream head table. The information printed is contained in an internal structure. The following members of this internal structure are described here:

address	Address of stream head
wq	Address of streams write queue
dev	Associated device number of the stream
read error	Read error on the stream
write_error	Write error on the stream
flags	Stream head flag values
push_cnt	Number of modules pushed on the stream
wroff	Write offset to prepend M_DATA
i**oc_id**	ID of outstanding M_IOCTL request
pollq	List of active polls
sigsq	List of active M_SETSIGs

The flags structure member, if set, is based on combinations of the following values:

#define	Value	Description
F_STH_READ_ERROR	0x0001	M_ERROR with read error received, fail all read calls.
F_STH_WRITE_ERROR	0x0002	M_ERROR with write error received, fail all writes.
F_STH_HANGUP	0x0004	M_HANGUP received, no more data.
F_STH_NDELON	0x0008	Do TTY semantics for ONDELAY handling.
F_STH_ISATTY	0x0010	This stream acts a terminal.
F_STH_MREADON	0x0020	Generate M_READ messages.
F_STH_TOSTOP	0x0040	Disallow background writes (for job control).
F_STH_PIPE	0x0080	Stream is one end of a pipe or FIFO.
F_STH_WPIPE	0x0100	Stream is the “write” side of a pipe.
F_STH_FIFO	0x0200	Stream is a FIFO.
F_STH_LINKED	0x0400	Stream has one or more lower streams linked.
F_STH_CTTY	0x0800	Stream controlling tty.
F_STH_CLOSED	0x4000	Stream has been closed, and should be freed.
F_STH_CLOSING	0x8000	Actively on the way down.

This subcommand can be issued from **crash** on either a running system or a system dump.

Aliases = **none**

```
> stream
ADDRESS   WRITEQ MAJ/MIN RERR WERR  FLAGS IOCID  WOFF PCNT POLQNEXT SIGQNEXT
59b1900  59c2a74  15,  0    0    0 0x0838     0     0    2         0         0
59d3c00  59d5a74  23,  5    0    0 0x0020     0     0    1         0         0
59ffe00  59ff074  23,  4    0    0 0x0020     0     0    1         0         0
5ab4c00  5ab4374  24,  0    0    5 0x0816     0     0    2         0         0
59d3f00  59ee974  24,  1    0    5 0x0816     0     0    2         0         0
59d3800  59dff74  24,  2    0    5 0x0816     0     0    2         0         0
59d3700  5a9c174  24,  3    0    5 0x0816     0     0    2         0         0
59ff800  59ff774  24,  4    0    0 0x0810     0     0    2         0         0
5a94d00  59ee574  24,  5    0    0 0x0830     0     0    2         0         0
5a94600  5a96c74  24,  6    0    5 0x0816     0     0    2         0         0
```

tcb [*ThreadTableEntry*] . . .

Displays the mstsave portion of the user structures of the named kernel threads (see the **user.h** and **mstsave.h** header files). If you do not specify an entry, information about the last running kernel thread is displayed. This subcommand replaces the **pcb** subcommand.

Aliases = none

```
> tcb
        UTHREAD AREA FOR SLOT   2
SAVED MACHINE STATE
  curid:0x00000204  m/q:0x00000000  iar:0x00019cfc  cr:0x22000000
  usr:0x00009030  lr:0x00035678  ctr:0x00019c90  xer:0x20000000
  *prevmst:0x00000000  *stackfix:0x00000000  intpri:0x0000000b
  backtrace:0x00  tid:0x00000000  fpeu:0x00  ecr:0x00000000
  Exception Struct
    0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
  Segment Regs
   0:0x00000000   1:0x007fffff   2:0x00000408   3:0x007fffff
   4:0x007fffff   5:0x007fffff   6:0x007fffff   7:0x007fffff
.
.
.
```

Aliases = **none**

```
>tcb

  UTHREAD AREA FOR SLOT  25

SAVED MACHINE STATE
    curid:0x0000162e  m/q:0x00000000  iar:0x0187e1dc  cr:0x44224820
    msr:0x000090b0  lr:0x0187e2d8  ctr:0x00040610  xer:0x00000004
    *prevmst:0x00000000  *stackfix:0x00000000  intpri:0x0000000b
    backtrace:0x00  tid:0x00000000  fpeu:0x01  ecr:0x00000000
    Exception Struct
      0xd0112540  0x42000000  0x400015b5  0xd0112540  0x00000106
    Segment Regs
     0:0x00000000   1:0x007fffff   2:0x000019f9   3:0x007fffff
     ...
    General Purpose Regs
     0:0x018abcd8   1:0x2fedefb8   2:0x018ac41c   3:0x018ab6e4
     4:0x018a63b0   5:0x018a63b0   6:0x018a63b0   7:0x00000000
     ...
    Floating Point Regs
        Fpscr: 0x00000000
     0:0x00000000 0x00000000  1:0x00000000 0x00000000  2:0x00000000 0x00000000
     3:0x00000000 0x00000000  4:0x00000000 0x00000000  5:0x00000000 0x00000000
     ...

Kernel stack address: 0x2fedf500

>tcb 10

  UTHREAD AREA FOR SLOT  10

SAVED MACHINE STATE
    curid:0x000009f4  m/q:0x00008003  iar:0x0001ddf8  cr:0x80222822
    msr:0x000010b0  lr:0x0001ddf8  ctr:0x000ee000  xer:0x00000000
    *prevmst:0x00000000  *stackfix:0x2fedf2d8  intpri:0x00000000
    backtrace:0x00  tid:0x00000000  fpeu:0x01  ecr:0x00000000
    Exception Struct
      0x30000000  0x40000000  0x00001272  0x30000000  0x00000106
    Segment Regs
     0:0x00000000   1:0x007fffff   2:0x00001010   3:0x007fffff
     ...
     General Purpose Regs
     0:0x40000707   1:0x2fedf2d8   2:0x00160d2c   3:0x00000420
     4:0x00000001   5:0xe6000644   6:0x000010b0   7:0x00000420
     ...
    Floating Point Regs
        Fpscr: 0x00000000
     0:0x00000000 0x00000000  1:0x00000000 0x00000000  2:0x00000000 0x00000000
     3:0x00000000 0x00000000  4:0x00000000 0x00000000  5:0x00000000 0x00000000
     ...

Kernel stack address: 0x2fedf500
```

thread [–] [–r] [–p *ProcessTableEntry* | –a *Address* | *ThreadTableEntry*]

Displays the contents of the kernel thread table. The **–** (minus) flag displays a longer listing of the thread table. The **–r** flag displays only runnable kernel threads. The **–p** flag displays only those kernel threads which belong to the process identified by *ProcessTableEntry*. The **–a** flag displays the kernel thread structure at *Address*. If *ThreadTableEntry* is given, only the corresponding kernel thread is displayed.

Aliases = **th**

```
> thread 1
SLT ST    TID      PID    CPUID  POLICY PRI CPU   EVENT  PROCNAME
  1 s     1e1        1  unbound   other  3c   0              init
        FLAGS: wakeonsig
```

The **trace** subcommand displays a kernel stack trace of the kernel thread identified by *ThreadTableEntry*. The trace starts at the bottom of the stack and attempts to find valid stack frames deeper in the stack. By default, the current kernel thread is used.

Use the **–r** flag to use the kernel frame pointer set up by the **kfp** subcommand as the starting address instead of the frame pointer found in the *SystemImageFile.* The **trace** subcommand stops and reports an error if an invalid frame pointer is encountered.

Aliases = **t**

```
> t 31
STACK TRACE:
        .et_wait ()
        .e_sleep ()
        .e_sleepl ()
        .sleepx ()
        .fifo_read ()
        .fifo_rdwr ()
        .vno_rw ()
        .rwuio ()
        .rdwr ()
        .kreadv ()
```

ts [*TextAddress*]

The **ts** subcommand finds the text symbols closest to the given address.

Aliases = **none**

```
> ts 012345
        .ioctl_systrace
```

tty [**d**] [**l**] [**e**] [*Name* | *Major* [*Minor*]]

Aliases = **term**, **dz**, **dh**

Refer to the **sys/tty.h** header file for the structure definition.

user [*ThreadTableEntry*]

Displays the uthread structure and the associated user structure of the thread identified by *ThreadTableEntry*. If you do not specify the entry, the information about the last running kernel thread is displayed..

Aliases = **u**, **uarea**, **u_area**

```
>u 4

     UTHREAD AREA FOR SLOT   4

SAVED MACHINE STATE
    curid:0x00000408  m/q:0x00008003  iar:0x0001ed98  cr:0x84201000
    msr:0x000010b0  lr:0x0001ed98  ctr:0x00000000  xer:0x20000000
    *prevmst:0x00000000  *stackfix:0x2feaeea8  intpri:0x00000000
    backtrace:0x00  tid:0x00000000  fpeu:0x00  ecr:0x00000000
    Exception Struct
      0x2feaf688  0x40000000  0x00000c0c  0x2feaf688  0x00000106
    Segment Regs
     0:0x00000000   1:0x007fffff   2:0x00000c0c   3:0x007fffff
     ...
    General Purpose Regs
     0:0x00000000   1:0x2feaeea8   2:0x00160d2c   3:0x00001000
     4:0x00000001   5:0x2fedf500   6:0x0000000b   7:0x000090b0
```

```
     ...
    Floating Point Regs
        Fpscr: 0x00000000
     0:0x00000000 0x00000000  1:0x00000000 0x00000000
     ...
    30:0x00000000 0x00000000 31:0x00000000 0x00000000

Kernel stack address: 0x2feaeffc

SYSTEM CALL STATE
    errno address:0xc0c0fade  error code:0x00  *kjmpbuf:0x00000000

PER-THREAD TIMER MANAGEMENT
    Real/Alarm Timer (ut_timer.t_trb[TIMERID_ALRM]) = 0x0
    Virtual Timer (ut_timer.t_trb[TIMERID_VIRTUAL]) = 0x0
    Prof Timer (ut_timer.t_trb[TIMERID_PROF]) = 0x0

SIGNAL MANAGEMENT
    *sigsp:0x0  oldmask:hi 0x0,lo 0x0  code:0x0

MISCELLANOUS FIELDS:
   fstid:0x00000000   ioctlrv:0x00000000   selchn:0x00000000

      USER AREA OF ASSOCIATED PROCESS gil (SLOT   4, PROCTAB 0xe3000400)
     handy_lock:0x00000000  timer_lock:0x00000000
    map:0x00000000  *semundo:0x00000000
    compatibility:0x00000000  lock:0x00000000

SIGNAL MANAGEMENT
    Signals to be blocked (sig#:hi/lo mask,flags,&func)
     1:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
     2:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
     3:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
     ...

USER INFORMATION
    euid:0x0000  egid:0x0000  ruid:0x0000  rgid:0x0000  luid:0x00000000
    suid:0x00000000  ngrps:0x0000  *groups:0x2feacc34  compat:0x00000000
    ref:0x00000004
    acctid:0x00000000   sgid:0x00000000   epriv:0x00000000
    ipriv:0x00000000   bpriv:0x00000000   mpriv:0x00000000

    u_info:

ACCOUNTING DATA
    start:0x2d612cc9  ticks:0x00000002  acflag:0x0000  pr_base:0x00000000
    pr_size:0x00000000  pr_off:0x00000000  pr_scale:0x00000000
    process times:
           user:0x00000000s 0x00000000us
            sys:0x000004f1s 0x14dc9380us
    children's times:
           user:0x00000000s 0x00000000us
            sys:0x00000000s 0x00000000us

CONTROLLING TTY
    *ttysid:0x00000000  *ttyp(pgrp):0x00000000
    ttyd(evice):0x00000000  ttympx:0x00000000  *ttys(tate):0x00000000
    tty id: 0x00000000  *query function: 0x00000000

PINNED PROFILING BUFFER
    *pprof: 0x00000000  *mem desc: 0x00000000

RESOURCE LIMITS AND COUNTERS
    ior:0x00000000  iow:0x00000000  ioch:0x00000000
```

```
    text:0x00000000  data:0x00000000  stk:0x01000000
    max data:0x08000000  max stk:0x01000000  max file:0x7fffffff
    soft core dump:0x7fffffff  hard core dump:0x7fffffff
    soft rss:0x7fffffff  hard rss:0x7fffffff
    cpu soft:0x7fffffff  cpu hard:0x7fffffff
    hard ulimit:0x7fffffff
    minflt:0x00000000   majflt:0x00000000

AUDITING INFORMATION
    auditstatus:0x00000000

SEGMENT REGISTER INFORMATION
    Reg        Flag      Fileno     Pointer
     0           0          0          0
     1           0          0          0
     ...
    *adspace:0xa0000000

FILE SYSTEM STATE
    *curdir:0x00000000  *rootdir:0x00000000
    cmask:0x0000  maxindex:0x0000
FILE DESCRIPTOR TABLE
    *ufd: 0x20013b14

> user
        UTHREAD AREA FOR SLOT  31
SAVED MACHINE STATE
  curid:0x00001fd8  m/q:0x00000000  iar:0x0006ee54  cr:0x2224248a
  msr:0x00009030  lr:0x000095d4  ctr:0x00000009  xer:0x00000020
  *prevmst:0x00000000  *stackfix:0x00000000  intpri:0x0000000b
  backtrace:0x00  tid:0x00000000  fpeu:0x01  ecr:0x00000000
  Exception Struct
    0x10013b7c  0x4000d030  0x60000990  0x10013b7c  0x00000106
  Segment Regs
   0:0x00000000   1:0x007fffff   2:0x0000068e   3:0x6000068e
   4:0x007fffff   5:0x007fffff   6:0x007fffff   7:0x007fffff
.
.
.
  21_trb[TIMERID_ALRM]) = 0x0
  Virtual Timer (ut_timer.t_trb[TIMERID_VIRTUAL]) = 0x0
  Prof Timer (ut_timer.t_trb[TIMERID_PROF]) = 0x0
SIGNAL MANAGEMENT
    *sigsp:0x0  oldmask:hi 0x0,lo 0x0  code:0x0
MISCELLANOUS FIELDS:
   fstid:0x00000000   ioctlrv:0x00000000   selchn:0x00000000
        USER AREA OF ASSOCIATED PROCESS crash (SLOT  31, PROCTAB
0xe3001f00)
    handy_lock:0x00000000  timer_lock:0x00000000
    map:0x00000000  *semundo:0x00000000
    compatibility:0x00000000  lock:0x00000000
SIGNAL MANAGEMENT
    Signals to be blocked (sig#:hi/lo mask,flags,&func)
     1:hi 0x00000000,lo 0x00000000,0x00000000,0x00000000
     2:hi 0x00x00000000
.
.
.
```

Refer to the **sys/user.h** header file for the structure definition.

var

The **var** subcommand displays the tunable system parameters.

Aliases = **tune**, **tunable**, **tunables**

```
> var
buffers       20
files        328
e_files      328
threads   262144
e_threads     51
clists     16384
maxproc       40
iostats        1
locks        200
e_locks  8456344
```

vfs [–] [*Vfs SlotNumber*]

The **vfs** uses the specified *Vfs SlotNumber* to display an entry in the **vfs** table. Use the **–** flag to display the vnodes associated with the **vfs**. The default displays the entire **vfs** table.

Aliases = **m**, **mnt**, **mount**

```
> vfs 3
VFS ADDRESS TYPE  OBJECT    STUB NUM FLAGS  PATHS
  3 1a62494  jfs 1a6d47c 1a6d650   5 D     /dev/hd1 mounted over /u
        flags:  C=disconnected D=device   I=remote P=removable
                R=readonly     S=shutdown U=unmounted Y=dummy
> vfs - 3
VFS ADDRESS TYPE  OBJECT    STUB NUM FLAGS  PATHS
  3 1a62494  jfs 1a6d47c 1a6d650   5 D     /dev/hd1 mounted over /u
ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT    INODE FLAGS
1a6e0ac   3    -   vreg    jfs     1     -  18f82c0
1a6e218   3    -   vreg    jfs     1     -  18f8770
1a6e24c   3    -   vreg    jfs     1     -  18f8590
1a6e17c   3    -   vdir    jfs     3     -  18f7f00
1a6dea4   3    -   vreg    jfs     2     -  18f65b0
1a6dfa8   3    -   vdir    jfs     5     -  18f6100
1a6d47c   3    -   vdir    jfs     1     -  18ea580 vfs_root
```

Refer to the **sys**/**vfs.h** header file for structure definitions.

vnode [*VNodeAddress*]

The **vnode** subcommand displays data at the specified *VNodeAddress* as a **vnode**. *VNodeAddress* must be specified in hexadecimal notation.The default is to display all **vnode**s in the **vnode** table.

Aliases = **none**

```
> vnode 1a6e078
ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT  DATAPTR FLAGS
1a6e078   0    -   vreg    jfs     4     -  18f6790
 Total VNODES printed 1
```

Refer to the **sys**/**vnode.h** header file for the structure definition.

xmalloc

The **xmalloc** subcommand displays information concerning the allocation and usage of kernel memory, specifically the **pinned_heap** and the **kernel_heap**.

Aliases = **xm**, **malloc**

```
>xmalloc

Kernel heap usage
heap size = 242720768 amount used = 79005568
Pinned heap usage
heap size = 242720768 amount used =   342832
Kernel and pinned heap usage
from =   1028ac bytes = 62914560 number =      2
from =    41d58 bytes =  8388608 number =      1

 ...
```

Kernel Debug Program

Use the kernel debug program (also known as the kernel debugger or low-level debugger) for debugging the kernel, device drivers, and other kernel extensions. The kernel debug program provides the following functions:

- Setting breakpoints within the kernel or within kernel extensions
- Formatting and displaying selected kernel data structures
- Viewing and modifying memory for any kernel data
- Viewing and modifying memory for kernel instructions
- Modifying the state of the machine by altering system registers

Loading and Starting the Kernel Debug Program

The kernel debug program must be loaded by using the **bosboot** command before it can be started. Use either of the following commands:

```
bosboot -a -d /dev/ipldevice -D
```

OR

```
bosboot -a -d /dev/ipldevice -I
```

The **–D** flag causes the kernel debugger program to be loaded. The **–I** flag also causes the kernel debug program to be loaded, but it is also invoked at system initialization. This means that when the system starts, it will trap the kernel debug program.

After issuing the **bosboot** command, you must restart the machine. The kernel debug program will not be loaded until the system is restarted. When started, the debug program accepts the commands described in “Kernel Debug Program Commands” on page 15-30.

If the kernel debug program is invoked during initialization, use the **go** command to continue the initialization process.

Note: The debug program disables all external interrupts while it is in operation.

Using a Terminal with the Kernel Debug Program

The debug program opens an asynchronous ASCII terminal when it is first started, and subsequently upon being started due to a system halt. Native serial ports are checked sequentially starting with port 0 (zero). Each port is configured at 9600 bps, 8 bits, and no parity. If carrier detect is asserted within 1/10 seconds, then the port is used. Otherwise, the next available native port is checked. This process continues until a port is opened or until every native port available on the machine has been checked. If no native serial port is opened successfully, then the result is unpredictable.

You can only display the kernel debugger on an ASCII terminal connected to a native serial port. The kernel debugger does *not* support any displays connected to any graphics adapters. The debugger has its own device driver for handling the display terminal. It is also possible to connect a serial line between two machines and define the serial line port as the port for the console. In that case, use the **cu** command to connect to the target machine and run the debugger.

Attention: If a serial device, other than a terminal connected to a native serial port, is selected by the kernel debugger, the system may appear to just hang up.

Entering the Kernel Debug Program

It is possible to enter the kernel debug program through one of the following procedures:

- From a native keyboard, press Ctrl-Alt-Numpad4.
- From the tty keyboard, enter Ctrl-4 (IBM 3151 terminals) or Ctrl-\ (BQ 303, BQ 310C, and WYSE 50).
- The system can enter the debugger if a breakpoint is set. To do this, use the **break** debugger command. See “Breakpoints” on page 15-29 and “Setting Breakpoints” on page 15-68 for information on setting a breakpoint.
- The system can also enter the debugger by calling the **brkpoint** subroutine from C code. The syntax for calling this subroutine is:

  ```
  brkpoint();
  ```

- The system can also enter the debugger if a static debug trap (SDT), is compiled into the code. To do this, place the assembler language instruction:

  ```
  t 0x4, r1 r1
  ```

  at the desired address. One way to do this is to create an assembler language routine that does this, then call it from your driver code.

  Note: After the debug program is started, SDTs are treated the same as other processor instructions. The **step** command can be used to step over SDTs. The **go** or **loop** commands can be used to resume execution at the instruction following the SDT.

- The system can also enter the debugger if a system halt is caused by a fatal system error. In such a case, the system creates a log entry in the system log and if the kernel debugger is available, calls the kernel debugger. A system dump is generated on exit from the debugger.

If the kernel debug program is not available (nothing happens when you type in the above key sequence), you must load it. To do this, see “Loading and Starting the Kernel Debug Program” on page 15-25.

Note: You can use the **crash** command to determine whether the kernel debug program is available. Use the **od** subcommand:

```
# crash
>od dbg_avail
```

If the **od** subcommand returns a 0 or 1, the kernel debug program is available. If it returns 2, the debug program is not available.

Debugging Multiprocessor Systems

On multiprocessor systems, entering the kernel debug program stops all processors (except the current processor running the debug program itself). Generally, when the debugger returns control to the program being debugged, other processors are released to run again. However, other processors are not released during the **step** command. On multiprocessor systems, the kernel debug program prompt indicates the current processor as follows:

>*ProcessorNumber*>

where *ProcessorNumber* identifies the current processor.

Kernel Debug Program Concepts

When the kernel debugger is invoked, it is the only running program. All processes are stopped, interrupts are disabled, and the cache is flushed. The system creates a new **mstsave** (machine state save) area for use by the debugger. However, the data displayed by the debugger comes from the **mstsave** area of the thread that was interrupted when the debugger was entered. After exiting from the kernel debugger, all the processes will continue to run unless you entered the debugger through a system halt.

Commands

The kernel debug program must be loaded and started before it can accept commands

Once in the kernel debugger, use the commands to investigate and make alterations. Each command has an alias or a shortened form. This is the minimum number of letters required by the debugger to recognize the alias as unique. See “Kernel Debug Program Commands” on page 15-30 for lists and descriptions of the commands.

Numeric Values and Strings

Numeric arguments are required to be hexadecimal for all commands except the **loop** and **step** commands and the slotnumber option of the **drivers** command, which all take a numeric count in decimal. Decimal numbers must either be decimal constants (0–9), variables, or expressions involving both options (see “Expressions” on page 15-29). Hexadecimal numbers can also include the letters A through F.

In some cases, only numeric constants are allowed. Wherever appropriate, this restriction is clearly identified.

On the other hand, a string is either a hexadecimal constant or a character constant of the form “*String*”. Hexadecimal constants can be no longer than 8 digits. Double quotation marks separate string constants from other data.

Variables

Variable names must start with a letter and can be up to eight characters long. Variable names cannot contain special symbols. Variables usually represent locations or values which are used again and again. A variable must not represent a valid number. Use the **set** command to define and initialize variables. Variables can contain from 1 to 4 bytes of numeric data or up to 32 characters of string data. You can release a variable with the **reset** command. You cannot use the **reset** command with reserved variables.

For example:

```
set name 1234           Sets your variable called name=1234
set s8 820c00e0         Sets seg reg 8 to point to the IOCC
```

Note that `s8` is a reserved variable.

Reserved Variables

There is a set of variables that have a reserved meaning for the kernel debug program. You can reference and change these variables, but they represent the actual hardware registers. There are also two variables (*fx* and *org*) reserved for use by the kernel debug program, which can be changed or set. If you change any registers while in the kernel debug program, the change remains in effect when you leave the kernel debug program. The reserved variables are:

bat0l BAT register 0, lower.

bat0u BAT register 0, upper.

bat1l BAT register 1, lower.

bat1u BAT register 1, upper.

bat2l BAT register 2, lower.

bat2u BAT register 2, upper.

cppr Current processor priority register.

cr Condition register.

ctr Count register.

dar Data address register.

dec Decrementer.

dsier Data storage interrupt error register.

dsisr Data storage interrupt status register.

eim0 External interrupt mask (low).

eim1 External interrupt mask (high).

eis0 External interrupt summary (low).

eis1 External Interrupt summary (high).

fp0–fp31 Floating point registers 0 through 31.

fpscr Floating point status and control register.

fx Address of the last item found by the **find** command.

iar Instruction Address Register (program counter). Points to the current instruction.

lr Link register.

mq Multiply quotient.

msr Machine State register.

org The current value of origin. It is useful to set this to the program load point.

peis0 Pending external interrupt status register 0.

peis1 Pending external interrupt status register 1.

r0 – r31 General Purpose Registers 0 through 31. These registers have the following usage conventions:

- **r0** Used on prologs. Not preserved across calls.
- **r1** Stack pointer. Preserved across calls.
- **r2** TOC. Preserved across calls.
- **r3 – r10** Parameter list for a procedure call. The first argument is r3, the second is r4 and so on until r10 is the 8th argument. These registers are not preserved across calls.
- **r11** Scratch. Pointer to FCN; DSA pointer to `int proc(env).`
- **r12** PL8 exception return. Value preserved across calls.
- **r13–r31** Scratch. Value preserved across calls.

rtcl Real Time clock (nano seconds).

rtcu Real Time clock (seconds).

s0–s15 Segment registers. If a segment register is *not* in use, it has a value of 007FFFFF.

sdr0 Storage description register 0.

sdr1 Storage description register 1.

sisr Data Storage-Interrupt Status register.

srr0	Machine status save/restore 0.
srr1	Machine status save/restore 1.
tbl	Time base register, lower.
tbu	Time base register, upper.
tid	Transaction register (fixed point).
xer	Exception register (fixed point).
xirr	External interrupt request register.

Expressions

The kernel debug program does not allow full expression processing. Expressions can only contain decimal or hex constants, variables and operators. The variable operators include:

+	addition
–	subtraction
*****	multiplication
/	division
>	dereference

The **>** operator indicates that the value of the preceding expression is to be taken as the address of the target value. The contents of the address specified by the evaluated expression are used in place of the expression.

You can enter expressions in the form *Expression(Expression)*. This form causes the two expressions to be evaluated separately and then added together. This form is similar to the base address syntax used in the assembler.

You can also enter expressions in the form *+Expression* or *–Expression.* This form causes the expression to be added to or subtracted from the origin (the reserved variable **org**.)

Expressions are processed from left to right only. The type of data specified must be the same for all terms in the expression.

Pointer Dereferences

A pointer dereference can be used to refer indirectly to the contents of a memory location. For example, assume that the 0xC50 location contains a counter. An expression of the form **c50>** can be used to refer to the counter. Any expression can be placed before the **>** (greater than) operator, including an expression involving another **>** operator. In this case multiple levels of indirection are used. To extend the example, if the FF7 location contains the C50 value, the expression **FF7>>** refers to the above counter.

The following examples show how to use a pointer dereference with the **alter** command:

```
alter 124> 0582
alter addrl>+8 d96e
```

In the first case, data is placed into the memory location pointed to by the word at the `124` address. The second case places the `d96e` variable into memory at the address computed by adding `8` to the word at the address in the **addrl** variable.

Breakpoints

The debugger creates a table of breakpoints that it internally maintains. The **break** command creates breakpoints. The **clear** command clears breakpoints. When the breakpoint is set, the debugger temporarily replaces the corresponding instruction with the trap instruction. The instruction overlaid by the breakpoint operates when you issue a **step** or **go** command.

A breakpoint can only be set if the instruction is not paged out. Breakpoints should not be set in any code used by the debugger.

For more information, see “Setting Breakpoints” on page 15-68.

Kernel Debug Program Commands

The following table shows the kernel debug program commands in alphabetical order:

Command	Alias	Description
alter	a	Alters memory.
back	b	Decrements the Instruction Address Register (IAR).
break	br	Sets a breakpoint.
breaks	breaks	Lists currently set breakpoints.
buckets	bu	Displays contents of **kmembucket** kernel structures.
clear	cl	Clears (removes) breakpoints.
cpu	cp	Sets the current processor or shows processor states.
display	d	Displays a specified amount of memory.
dmodsw	dm	Displays the STREAMS driver switch table.
drivers	dr	Displays the contents of the device driver (**devsw**) table.
find	f	Finds a pattern in memory.
float	fl	Displays the floating point registers.
fmodsw	fm	Displays the STREAMS module switch table.
go	g	Starts the program running.
? or help	h	Displays the list of valid commands.
loop	l	Run until control returns to this point.
map	m	Displays the system loadlist.
mblk	mb	Displays the contents of message block structures.
next	n	Increments the IAR.
origin	o	Sets the origin.
ppd	pp	Displays per-processor data.
proc	pr	Displays the formatted process table.
queue	que	Displays contents of STREAMS queue at specified address.
quit	q	Ends a debugging session.
reset	r	Releases a user-defined variable.
screen	s	Displays a screen containing registers and memory.
set	se	Defines or initialize a variable.
sregs	sr	Displays segment registers.
st	st	Stores a fullword in memory.
stack	sta	Displays a formatted kernel stack trace.
stc	stc	Stores one byte in memory.
step	ste	Performs an instruction single-step.
sth	sth	Stores a halfword in memory.
stream	str	Displays stream head table.

Command	Alias	Description
swap	sw	Switches from the current display and keyboard to another RS232 port.
thread	th	Displays thread table entries.
trace	tr	Displays formatted trace information.
trb	trb	Displays the timer request blocks.
tty	tt	Displays the **tty** structure.
user	u	Displays a formatted user area.
uthread	ut	Displays the **uthread** structure.
vars	v	Displays a listing of the user-defined variables.
vmm	vm	Displays the virtual memory data structure.
xlate	x	Translates a virtual address to a real address.

Kernel Debug Program Commands Grouped by Task Categories

The kernel debug program commands can be grouped into the following task categories:

- Displaying Registers
- Modifying Registers
- Setting, Specifying, and Deleting Breakpoints
- Displaying Data
- Manipulating Memory
- Controlling the Debugger

Displaying Registers

cpu Selects the current processor.

float Displays the floating-point register.

origin Sets the origin of the IAR.

screen Displays a screen containing registers and memory.

sregs Displays segment registers.

Modifying Registers

back Decreases the instruction address register (IAR).

next Increments the IAR.

set Define or initialize a user-defined variable.

Setting, Specifying, and Deleting Breakpoints

brat Sets a branch on target address (brat) point.

break Sets a breakpoint.

breaks Lists currently set breakpoints.

clear Removes breakpoints.

go Starts the operation of the program following a breakpoint or static debug trap.

loop Operates until control returns to this point a number of times.

step Performs a single-step instruction.

watch Sets watch points which interrupt data storage.

Displaying Data

buckets Displays statistics on the *net_malloc* kernel memory pool by bucket size.

display Displays a specified amount of memory.

dmodsw Displays the internal STREAMS driver switch table.

drivers Displays the contents of the device driver (devsw) table.

fmodsw Displays the internal STREAMS module switch table.

map Displays a system load list.

mblk Displays the contents of the STREAMS message blocks.

ppd Displays a formatted per-processor data structure.

proc Displays the formatted process table.

queue Displays the contents of the STREAMS queues.

screen	Displays a screen containing registers and memory.
stack	Displays a formatted kernel stack trace.
stream	Displays the contents of the stream head table.
thread	Displays the formatted thread table.
trace	Displays formatted trace information.
trb	Displays the timer request blocks.
tty	Displays **tty** information.
user	Displays a formatted user area.
uthread	Displays a formatted **uthread** structure.
vmm	Displays the virtual memory information menu.

Manipulating Memory

alter	Alters memory.
display	Displays a specified amount of memory.
find	Finds a pattern in memory.
st	Stores a fullword in memory.
stc	Stores 1 byte in memory.
sth	Stores a halfword in memory.
vmm	Displays the virtual memory information menu.
xlate	Translates a virtual address to a real address.

Controlling the Debugger

? or help	Displays the list of valid commands.
quit	Ends the debugging session.
reset	Clear a user-defined variable.
set	Define or initialize a user-defined variable.
swap	Switches from the current display and keyboard to an RS-232 port.
vars	Displays a listing of user-defined variables.

Descriptions of the Kernel Debug Program Commands

This includes a description of each of the kernel debug program commands. The commands are in alphabetical order.

alter Command for the Kernel Debug Program

Purpose

Alters a memory location to the hexadecimal value entered.

Syntax

Description

The **alter** command changes the memory location specified by the *Address* parameter to the hexadecimal value specified by the *Data* parameter. The **alter** command can be used to change one or several bytes of memory. The number of bytes modified with this command depends on the number of bytes you specified. If you specified an odd number of hexadecimal digits, only the first four bits of the last byte are changed.

The **alter** command cannot be used to modify storage to the value of a variable or an expression. Instead, use the **st** command, the **stc** command, or the **sth** command.

Examples

1. To store the 16-bit `ffff` value at the 1000 address, enter:

   ```
   alter 1000 ffff
   ```

2. To store the 8-bit 2C value in the high-order byte at the 1000 address, enter:

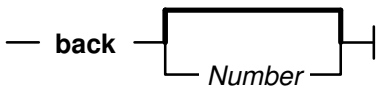
   ```
   a 1000 2C
   ```

back Command for the Kernel Debug Program

Purpose

Decreases the instruction address register (IAR).

Syntax



Description

The **back** command decreases the IAR by the number of bytes specified by the *Number* parameter and displays the new current instruction.

Examples

1. To decrement the IAR by 4 bytes, enter:

   ```
   back
   ```

2. To decrement the IAR by 16 bytes, enter:

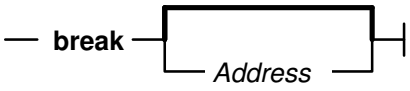
   ```
   b 16
   ```

break Command for the Kernel Debug Program

Purpose

Sets a breakpoint.

Syntax



Description

The **break** command sets a breakpoint in a program at the address specified by the *Address* parameter. The *Address* parameter should be a hexadecimal expression. A breakpoint starts the loaded debug program when the instruction at the specified address is run.

There is a maximum of 32 breakpoints.

Examples

1. To set a breakpoint at the instruction address register (IAR), enter:

   ```
   break
   ```

2. To set a breakpoint at address 521A, enter:

   ```
   break 521a
   ```

3. To set a breakpoint at A0+8300, enter:

   ```
   br 8300+A0
   ```

4. To set a breakpoint at the origin plus A0, enter:

   ```
   break +A0
   ```

5. To set a breakpoint at the address in the link register, enter:

   ```
   break lr
   ```

breaks Command for the Kernel Debug Program

Purpose

Lists the current breakpoints.

Syntax

Description

The **breaks** command lists all currently active breakpoints. For each breakpoint, an offset into a segment is given along with the segment register value at the time the breakpoint was set. This information is required to distinguish between breakpoints set at identical offsets from different segment register values.

buckets Command for the Kernel Debug Program

Purpose

Displays statistics on the *net_malloc* kernel memory pool by bucket size.

Syntax

Description

The **buckets** command displays the contents of the **kmembucket** kernel structures. These structures contain information on the *net_malloc* memory pool by size of allocation.

All output values are printed in hexadecimal format.

This command can also be invoked via the alias, **bu**.

Example

To display **kmembucket** kernel structure for offset 0 and allocation size of 2 enter:

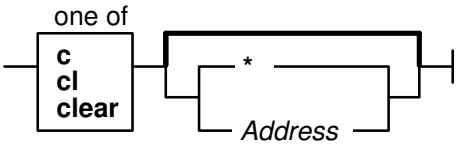
```
buckets
```

clear Command for the Kernel Debug Program

Purpose

Removes one or all breakpoints.

Syntax



Description

The **clear** command removes one or all breakpoints. The *Address* parameter specifies the location of the breakpoint to be removed. If you specify no flags, the breakpoint pointed to by the instruction address register (IAR) is removed. The **clear** command can be initiated by entering `clear`, `c`, or `cl` at the command line.

Addresses are maintained as offsets from the start of their segment. In the event that two breakpoints are set at the same offset at the start of two different segments, and one breakpoint is then removed, the address specified to the **clear** command is not unique. In this case, each of the conflicting segment IDs are displayed, and the **clear** command displays a prompt requesting the ID of the segment whose breakpoint you want to remove.

Examples

1. To clear the breakpoint at the IAR, enter:

   ```
   clear
   ```

2. To clear the breakpoint at the 10000200 address, enter:

   ```
   cl 10000200
   ```

3. To clear all breakpoints, enter:

   ```
   clear *
   ```

cpu Command for the Kernel Debug Program

Purpose

Switches the current processor, and reports the kernel debug state of processors.

Syntax

Description

The **cpu** command places the processor specified by the *ProcessorNumber* parameter in debug mode; the processor enters the debugger and is ready to accept commands. The processor where the debugger was previously running is stopped. This command is available only on multiprocessor systems.

If no processor is specified, the **cpu** command displays the kernel debug state of each processor. The possible states are as follows:

`Debug`	The processor has entered the debugger.
`Stopped`	The processor has been stopped by another processor in the debug state.
`Waiting`	The processor has hit a breakpoint while another processor is in the debug state, without having been stopped by the other processor. A particular example is the race condition where two processors both hit breakpoints. One of the processors will enter the debug state; the other will enter the waiting state.

Example

To select the first processor, enter:

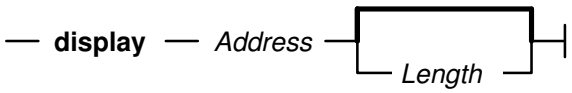
```
cpu 0
```

display Command for the Kernel Debug Program

Purpose

Displays a specified amount of memory.

Syntax



Description

The **display** command displays memory storage, starting at the address specified by the *Address* parameter. The *Length* parameter indicates the number of bytes to display, and has a default value of 16.

The **display** command displays the contents of the specified region of memory in a two-column format. The left column displays the contents of memory in hexadecimal, and the right column displays the printable ASCII representation of the hexadecimal data.

The **display** command also shows the exact amount of storage requested when you specify a length of 1, 2, or 4 bytes. In this instance, it uses the processor load character, load halfword, or load fullword instruction, respectively. These instructions should be used when displaying input and output address space. Any other value for the *Length* parameter causes memory to be loaded one byte at a time.

Examples

1. To display `16` bytes at the `IAR`, enter:

   ```
   display iar
   ```

2. To display `12` bytes at address `152F`, enter:

   ```
   d 152F 12
   ```

3. To display `16` bytes at the origin `+ B7`, enter:

   ```
   display +B7
   ```

4. To display `16` bytes at the address in `r3`, enter:

   ```
   disp r3
   ```

5. To display from the address contained in the address in `r3`, enter:

   ```
   d r3>
   ```

dmodsw Command for the Kernel Debug Program

Purpose

Displays the internal STREAMS driver switch table.

Syntax

dmodsw

Description

The **dmodsw** command displays the internal STREAMS driver switch table, one entry at a time. By pressing the Enter key, you can walk through all the **dmodsw** entries in the table. The contents of the first entry are meaningless except for the *d_next* pointer. When the last entry has been reached, the **dmodsw** command will print the message, "`This is the last entry.`"

The information printed is contained in an internal structure. The following members of this internal structure are described here:

address	Address of dmodsw
d_next	Pointer to the next driver in the list
d_prev	Pointer to the previous driver in the list
d_name	Name of the driver
d_flags	Flags specified at configuration time
d_sqh	Pointer to synch queue for driver–level synchronization
d_str	Pointer to streamtab associated with the driver
d_sq_level	Synchronization level specified at configuration time
d_refcnt	Number of open or pushed count
d_major	Major number of a driver

The *flags* structure member, if set, is based one of the following values:

```
#define            value  description
----------------   -----  ----------------------------------------
F_MODSW_OLD_OPEN     0x1  Supports old-style (V.3) open/close
                          parameters
F_MODSW_QSAFETY      0x2  Module requires safe timeout/bufcall
                          callbacks
F_MODSW_MPSAFE       0x4  Non-MP-Safe drivers need funneling
```

The synchronization level codes are described in the **/usr/include/sys/strconf.h** header file.

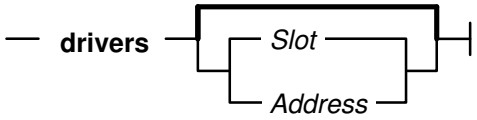
This command can also be invoked via the alias, **dm**.

drivers Command for the Kernel Debug Program

Purpose

Displays the contents of the device driver (**devsw**) table.

Syntax



Description

The **drivers** command displays the contents of the **devsw** table. If no parameters are specified, then each entry in the table is displayed. If a parameter is specified and is a valid slot number (less than 256), then the corresponding slot in the **devsw** table is displayed. If the parameter is not a valid slot number, then it is understood as an address and the slot with the last entry point prior to the given address is displayed, along with the name of that entry point.

Each **devsw** entry consists of a number of entry points (read, write, and so on) into the specified driver. Each entry consists of a function descriptor, and the address of the function.

Examples

1. To display the entire **devsw** table, enter:

   ```
   drivers
   ```

2. To display the tenth slot of the **devsw** table, enter:

   ```
   drivers 10
   ```

3. To display the last entry point before the address 0x130000F, enter:

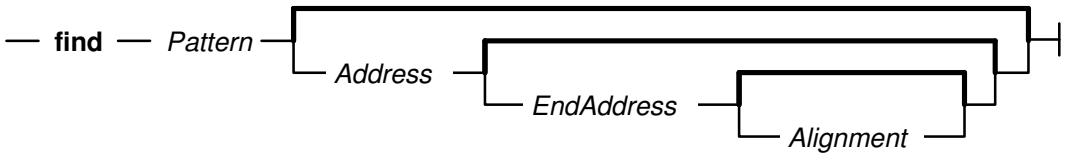
   ```
   dr 130000f
   ```

find Command for the Kernel Debug Program

Purpose

Searches storage.

Syntax



Description

The **find** command searches storage for a pattern beginning at the address specified by the *Address* parameter. If the specified argument is found, the search stops and storage containing the specified argument is displayed. The address of the storage is placed into the **fx** variable.

The following defaults apply to the first execution of the **find** command:

- *Address* = 0
- *EndAddress* = 0xFFFFFFFF
- *Alignment* = 1 (byte alignment)

An asterisk (*) can be substituted for any of the parameters. An asterisk causes the **find** command to use the value for that parameter that was used in the previous execution of the command.

Examples

1. To find the first occurrence of `7c81` in virtual memory starting at 0, enter:

   ```
   find 7c81
   ```

2. To find the first occurrence of the string `TEST`, enter:

   ```
   find "TEST"
   ```

3. To find the first occurrence of `7c81` after address `10000`, enter:

   ```
   f 7c81 10000
   ```

4. To find the first occurrence of `7c81` between `0` and the user-defined `top` variable, enter:

   ```
   f 7c81 0 top
   ```

5. To find the first occurrence of `7c81` starting at the last address used, enter:

   ```
   find 7c81 *
   ```

6. To find the first of occurrence of `7c81` starting at the last address used and aligned on a halfword, enter:

   ```
   f 7c81 * * 2
   ```

7. To find the next occurrence of `7c` starting at `1` plus the last address at which the **find** command stopped, enter:

   ```
   f 7c fx+1 * 2
   ```

8. To search for the last pattern used, enter:

   ```
   find *
   ```

9. To search for the last pattern starting at the next location (the **find** command remembers the alignment which was used in the previous search), enter:

   ```
   f * fx+1
   ```

float Command for the Kernel Debug Program

Purpose

Displays floating-point registers.

Syntax

Description

The **float** command displays the contents of floating-point registers and other control registers.

fmodsw Command for the Kernel Debug Program

Purpose

Displays the internal STREAMS module switch table.

Syntax

Description

The **fmodsw** command displays the internal STREAMS module switch table, one entry at a time. By pressing the Enter key, you can walk through all the **fmodsw** entries in the table. The contents of the first entry are meaningless except for the *d_next* pointer. When the last entry has been reached, the **fmodsw** command will print the message `This is the last entry`. This command can also be invoked via the alias, **fm**.

The information printed is contained in an internal structure. The following members of this internal structure are described here:

address	Address of **fmodsw**
d_next	Pointer to the next module in the list
d_prev	Pointer to the previous module in the list
d_name	Name of the module
d_flags	Flags specified at configuration time
d_sqh	Pointer to synch queue for module–level synchronization
d_str	Pointer to streamtab associated with the module
d_sq_level	Synchronization level specified at configuration time
d_refcnt	Number of open or pushed count
d_major	–1

The *flags* structure member, if set, is based one of the following values:

```
#define            value  description
----------------   -----
-------------------------------------------
F_MODSW_OLD_OPEN     0x1  Supports old-style (V.3) open/close
                          parameters
F_MODSW_QSAFETY      0x2  Module requires safe timeout/bufcall
                          callbacks
F_MODSW_MPSAFE       0x4  Non-MP-Safe drivers need funneling
```

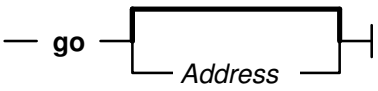
The synchronization level codes are described in the **/usr/include/sys/strconf.h** header file.

go Command for the Kernel Debug Program

Purpose

Starts executing the program under test.

Syntax



Description

The **go** command resumes operation of your program. Program operation begins at the current instruction address register (IAR) setting. Specify an address with the *Address* parameter to set the Instruction Address Register (IAR) to a new address and begin running there. If this command is used with no parameters after the debugger was entered via a fatal system error, a system dump will be generated and the machine will halt.

Examples

1. To continue running your program at the IAR, enter:

   ```
   go
   ```

2. To set the IAR to 1000 and begin running there, enter:

   ```
   g 1000
   ```

help Command for the Kernel Debug Program

Purpose

Displays the help screen of the kernel debug program.

Syntax

Description

The **help** command displays a one-line help message for each debug program command.

Example

To display the list of valid kernel debug program commands, enter:

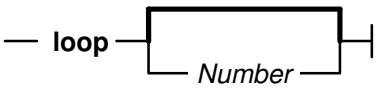
```
help
```

loop Command for the Kernel Debug Program

Purpose

Runs the program being tested until the IAR reaches the current value several times.

Syntax



Description

The **loop** command causes the system to continue running and to stop when the instruction address register (IAR) returns to the current value the number of times specified by the *Number* parameter. All other breakpoints are ignored. The *Number* parameter specifies the number of loops that execute before the debug program regains control, and must be a valid decimal expression. The default value for the *Number* parameter is 1.

The **loop** command is similar to setting a breakpoint at the current IAR, but allows you to stop on a specified instance when the IAR returns to the current point.

Example

To execute until the second time the IAR has the current value, enter:

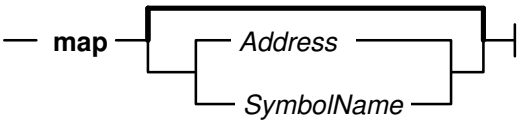
```
loop 2
```

map Command for the Kernel Debug Program

Purpose

Displays the system load list.

Syntax



Description

The **map** command displays information from the system load list. The system load list is the list of symbols exported from the kernel. If the **map** command is entered with no parameters, then the entire load list is displayed one page at a time. If an address is given, the name and value of the last symbol located before the given address is displayed. If a symbol name is given, then the load list is searched for the symbol and any matching entries are displayed. There can be more than one entry for a given symbol table.

Since the load list contains only symbols exported from the kernel, a given symbol name can be in the kernel but not reported by the **map** command.

The symbol value for a data structure is the address of that data structure. The symbol value for a function is not the address of the function, but the address of the function descriptor. The first word of the function descriptor is the address of the function. For example, if entering `map execexit` displays `0x1000`, then entering `display 1000` displays the address of the `execexit` function in the first word of the displayed memory.

Examples

1. To display the entire load list, enter:

   ```
   map
   ```

2. To display the symbol with a value closest to 0xe3000000, enter:

   ```
   m e3000000
   ```

3. To display the value of the function `execexit`, enter:

   ```
   map execexit
   ```

mblk Command for the Kernel Debug Program

Purpose

Displays the contents of the STREAMS message blocks defined by the **msgb** structure in the /**usr**/**include**/**sys**/**stream.h** header file.

Syntax

Description

The **mblk** command displays the contents of the **msgb** structure that is defined in the /**usr**/**include**/**sys**/**stream.h** headerfile. If you do not specify an *Address*, the command displays the contents of the message blocks of type **M_MBLK** and **M_MBDATA**, as well as displays the address of *mh_freelater*.

The *mh_freelater* parameter is a pointer to the message blocks that are just now freed and are scheduled to be given back to the system, but are not yet given back.

All output values are printed in hexadecimal format.

This command can also be invoked via the alias, **mb**.

Examples

1. To display the contents of the message blocks of type **M_BLK** and **M_MBDATA**, and the address of *mh_freelater*, enter:

   ```
   mblk
   ```

2. To display the contents of the message block structure at address `0005ec80`, enter:

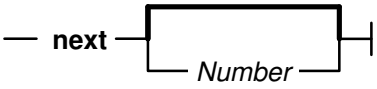
   ```
   mblk 0005ec80
   ```

next Command for the Kernel Debug Program

Purpose

Increases the instruction address register (IAR).

Syntax



Description

The **next** command increases the instruction address register (IAR) by the number specified by the *Number* parameter and displays the new current instruction. The default value for the *Number* parameter is 4 bytes.

Examples

1. To increment the IAR by 4 bytes, enter:

   ```
   next
   ```

2. To increment the IAR by 20 bytes, enter:

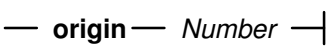
   ```
   n 20
   ```

origin Command for the Kernel Debug Program

Purpose

Sets the address origin of the instruction address register (IAR).

Syntax



Description

The **origin** command sets the address origin. The origin address specified by the *Number* parameter is added to any hexadecimal expression beginning with a + (plus sign). This command is especially useful when setting breakpoints. Use the **screen** command to display the value of the origin and the origin displacement of the IAR.

The **origin** command also sets the reserved **org** variable. For example, entering `origin 652C0` does the same as entering `set org 652C0`.

Examples

1. To set the origin to 178D, enter:

   ```
   origin 178D
   ```

2. To set the origin to 59cc, enter:

   ```
   o 59cc
   ```

ppd Command for the Kernel Debug Program

Purpose

Displays per-processor data.

Syntax

Description

The **ppd** command displays the per-processor data structure of the specified processor. If no argument is given, data for the current processor, as selected by the **cpu** command, is displayed.

Note: The **ppd** command is available only on multiprocessor systems.

Examples

1. To display per-processor data for the current processor, enter:

   ```
   ppd
   ```

2. To display per-processor data for processor 2, enter:

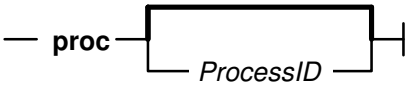
   ```
   ppd 2
   ```

proc Command for the Kernel Debug Program

Purpose

Displays the formatted process table.

Syntax



Description

The **proc** command displays the process table in a format similar to the output of the **ps** command, with an ***** (asterisk) placed next to the currently running process on the processor where the debugger is active. If the *ProcessID* parameter is specified, the **proc** command displays information pertaining to this process only, and gives more detailed information.

Examples

1. To display the process table, enter:

   ```
   p
   ```

2. To display the process table entry for the process with process ID 1, enter:

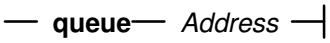
   ```
   proc 1
   ```

queue Command for the Kernel Debug Program

Purpose

Displays the contents of the STREAMS queues.

Syntax



Description

The **queue** command displays the contents of the STREAMS queue at the specified *Address*. Refer to the **/usr/include/sys/stream.h** header file for the queue structure definition.

In the output, an `X` indicates that the value is printed in hexadecimal format.

This command can also be invoked via the alias, **que**.

Example

To display the contents of the STREAMS queue stored at address `59c1874`, where `59c1874` is a valid queue address, enter:

```
queue 59c1874
```

quit Command for the Kernel Debug Program

Purpose

Ends the debug program session.

Syntax

Description

The **quit** command terminates the debug session. Use this command when you have completed debugging and want to clear all breakpoints. The **quit** command performs the following tasks:

- Clears all breakpoints.
- Issues the **go** command, which generates a system dump if the debugger was entered via a fatal system error.

To use the debug program again after issuing the **quit** command, use one of the keyboard sequences described in “Entering the Kernel Debug Program” on page 15-26.

reset Command for the Kernel Debug Program

Purpose

Clears a user-defined variable.

Syntax

Description

The **reset** command clears those variables specified with the *VariableName* parameter. Resetting a variable effectively deletes it, and allows the variable slot to be used again. Currently, 16 user-defined variables are allowed, and when they are all in use, you cannot set any more. Use the **vars** command to display all variables currently set.

Variables that are not user-defined, such as registers, cannot be reset. If you specify a variable that is not user-defined, or a variable that is not defined, an error message is displayed.

Example

To delete the user-defined variable **foo**, enter:

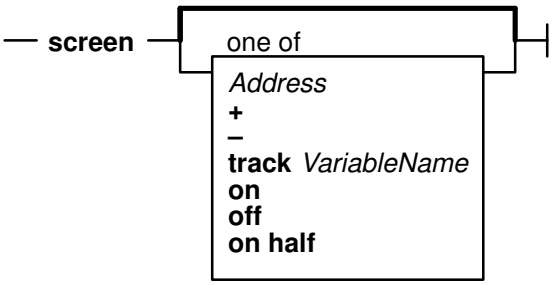
```
reset foo
```

screen Command for the Kernel Debug Program

Purpose

Displays a screen of data.

Syntax



Description

The **screen** command primarily displays memory and registers, but it is also used to control the format of subsequent **screen** commands. By default, memory is displayed starting at the instruction address register (IAR), or at the variable currently tracked. Variables can be tracked by specifying them with the **track** *VariableName* flag.

The track option changes the address that the screen displays as the expression that is being tracked changes. This option is useful in a case where, at a breakpoint, the memory to be displayed is addressed by a register.

You can also use parameters to modify the format of the screen so that only half of the physical screen is used, or even turn off the screen display entirely. The format modification parameters are useful if important information can be scrolled off the screen when the debugger is entered. Restore the default (full) screen by entering:

```
screen on
```

Flags

+	Displays the next 0x70 bytes of data.
–	Displays the previous 0x70 bytes of data.
track *VariableName*	Instructs the screen display to track to the specified variable.
on	Turns the display on.
off	Turns the display off so that the screen display does not appear when the debug program is started. This flag is useful if a slow, asynchronous terminal is used.
on half	Displays only the top half of the display screen. The memory display is omitted.

Examples

1. To display the next 112 bytes of data, enter:

   ```
   screen +
   ```

2. To display the previous 112 bytes of data, enter:

   ```
   screen –
   ```

3. To display memory starting at 20000FF7, enter:

   ```
   s 20000ff7
   ```

4. To display memory at the address contained in location 200, enter:

   ```
   s 200>
   ```

5. To turn on the display, enter:

   ```
   screen on
   ```

6. To turn off the display, enter:

   ```
   screen off
   ```

7. To set the display format to use about half of the screen, enter:

   ```
   screen on half
   ```

8. To track memory starting at the value in general purpose register 3, enter:

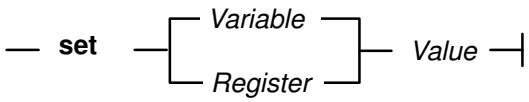
   ```
   sc track r3
   ```

set Command for the Kernel Debug Program

Purpose

Create and change values of debugger variables.

Syntax



Description

This command sets debugger variables. Use the **set** command to create new variables or modify the value of old variables. Certain debugger variables are symbolic names for machine registers, which you can modify. See “Reserved Variables” on page 15-27 for a list of these variables.

Examples

1. To assign value 100 to variable **start**.

   ```
   set start 100
   ```

2. To set general purpose register 12 to 0.

   ```
   set r12 0
   ```

3. To set segment register 3 to 10000.

   ```
   se s3  10000
   ```

4. To assign 45F0 to the Iar.

   ```
   set iar 45F0
   ```

5. To assign string `"AIX"` to variable **name**.

   ```
   se name "AIX"
   ```

sregs Command for the Kernel Debug Program

Purpose

Displays segment registers.

Syntax

Description

The **sregs** command displays the contents of the segment registers and other control registers. The display created is similar to that created by the **screen** command.

st Command for the Kernel Debug Program

Purpose

Stores a fullword into memory.

Syntax

Description

The **st** command stores a fullword of data into memory by using the processor fullword store instruction. If the address specified by the *Address* parameter is not word-aligned, it is rounded down to a fullword. The **st** command is the correct way to place a fullword of data into input and output memory.

This is similar to the **alter** command, but the word size is implicit in the command. **stc** and **sth** are used to perform similar functions for bytes and halfwords.

Example

To store the 32-bit value `5` at address `1000`, enter:

```
st 1000 5
```

stack Command for the Kernel Debug Program

Purpose

Displays a formatted stack traceback.

Syntax

Description

The **stack** command displays a formatted kernel-stack traceback for the specified kernel thread. If no thread is specified, the currently running thread is used. Stack frames show return addresses and can be used to trace the calling sequence of the program. Be aware that the first few parameters are passed in registers to the called functions, and are not usually available on the stack. Generally only the stack chain (stacks back-chain pointer) and return address (address where the current function returns upon completion) are valid. To interpret the stack thoroughly, it is necessary to use an assembler language listing for a procedure to determine what has been stored on the stack. Stack frames for the specified thread are not always accessible.

Examples

1. To format any existing stack frames, enter:

   ```
   stack
   ```

2. To format stack frames for the thread with thread ID 251 enter:

   ```
   sta 251
   ```

stc Command for the Kernel Debug Program

Purpose

Stores one byte into memory.

Syntax

Description

The **stc** command stores a byte of data specified by the *Data* parameter into memory at the address specified by the *Address* parameter by using the processor store-character instruction. The **stc** command is the correct way to place a byte of data into input and output memory.

This is similar to the **st** and **sth** commands, which are used for fullwords and halfwords.

Example

To store the 8-bit value `FF` at address `1000`, enter:

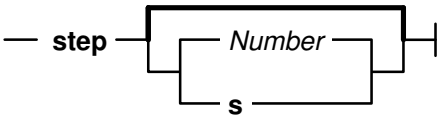
```
stc 1000 ff
```

step Command for the Kernel Debug Program

Purpose

Runs instructions single-step.

Syntax



Description

The **step** command causes the processor to enter a single instruction and return control to the debug program. If a branch is the next instruction to be run, the **s** flag causes the processor to step over a subroutine call. An integer *Number* parameter is used as the number of instructions to run before returning control to the debug program.

Note: On multiprocessor systems, other processors are not released during **step**, contrary to most commands.

Flag

s	Executes a subroutine as if it were one instruction.

Examples

1. To single step the processor, enter:

   ```
   step
   ```

2. To single step and skip over a subroutine call, enter:

   ```
   step s
   ```

3. To step for 20 instructions, enter:

   ```
   step 20
   ```

sth Command for the Kernel Debug Program

Purpose

Stores a halfword into memory.

Syntax

Description

The **sth** command stores a halfword of data specified by the *Data* parameter into memory by using the processor store halfword instruction. If the address specified by the *Address* parameter is not halfword-aligned, it is rounded down to a halfword boundary. The **sth** command is the correct way to place a halfword into input and output memory space.

This is similar to the **st** and **stc** commands, which are used for fullwords and bytes.

Example

To store the 16-bit value `14` at address `1000`, enter:

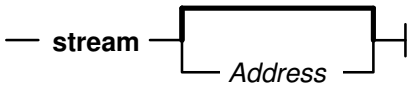
```
sth 1000 0014
```

stream Command for the Kernel Debug Program

Purpose

Displays the contents of the stream head table.

Syntax



Description

The stream command displays the contents of the stream head table. If no address is specified, the command displays the first stream found in the STREAMS hash table. If the address is specified, the command displays the contents of the stream head stored at that address.

The information printed is contained in an internal structure. The following members of this internal structure are described here:

sth	address of stream head
wq	address of streams write queue
rq	address of streams read queue
dev	associated device number of the stream
read_mode	read mode
write_mode	write mode
close_wait_timeout	close wait timeout in microseconds
read_error	read error on the stream
write_error	write error on the stream
flags	stream head flag values
push_cnt	number of modules pushed on the stream
wroff	write offset to prepend M_DATA
ioc_id	id of outstanding M_IOCTL request
*ioc_m*p	outstanding ioctl message
next	next stream head on the link
pollq	list of active polls
sigsq	list of active M_SETSIGs
shttyp	pointer to tty information

The *read_mode* and *write_mode* values are defined in the **/usr/include/sys/stropts.h** header file.

The *read_error* and *write_error* variables are integers defined in the **/usr/include/sys/errno.h** header file.

The *flags* structure member, if set, is based on combinations of the following values:

#define	Value	Description
F_STH_READ_ERROR	0x0001	M_ERROR with read error received, fail all read calls.
F_STH_WRITE_ERROR	0x0002	M_ERROR with write error received, fail all writes.
F_STH_HANGUP	0x0004	M_HANGUP received, no more data.
F_STH_NDELON	0x0008	Do TTY semantics for ONDELAY handling.
F_STH_ISATTY	0x0010	This stream acts a terminal.
F_STH_MREADON	0x0020	Generate M_READ messages.
F_STH_TOSTOP	0x0040	Disallow background writes (for job control).
F_STH_PIPE	0x0080	Stream is one end of a pipe or FIFO.
F_STH_WPIPE	0x0100	Stream is the “write” side of a pipe.
F_STH_FIFO	0x0200	Stream is a FIFO.
F_STH_LINKED	0x0400	Stream has one or more lower streams linked.
F_STH_CTTY	0x0800	Stream controlling tty.
F_STH_CLOSED	0x4000	Stream has been closed, and should be freed.
F_STH_CLOSING	0x8000	Actively on the way down.

In the output, values marked with `X` are printed in hexadecimal format.

This command can also be invoked via the alias, **str**.

Examples

1. To display the first stream head found in the stream head table, enter:

   ```
   stream
   ```

2. To display the contents of the particular stream head located at address `59b2e00` (where `59b2e00` is a valid stream head address), enter:

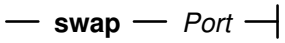
   ```
   stream 59b2e00
   ```

swap Command for the Kernel Debug Program

Purpose

Switches to the specified RS-232 port.

Syntax



Description

The **swap** command allows control of the debug program to be transferred to another terminal. The *Port* parameter specifies which asynchronous tty port to transfer control. The **swap** command does not support returning to a port that was previously used.

Specify 0 for port 0 (s1) or 1 for port 1 (s2).

Ports must be configured the same as the port on which the debug program is currently running: 9600 baud, 8 data bits, no parity. The device attached to the port must respond with a carrier detect within 1/10 seconds or the command fails and control will not be transferred.

Example

To switch display to RS-232 port `1`, enter:

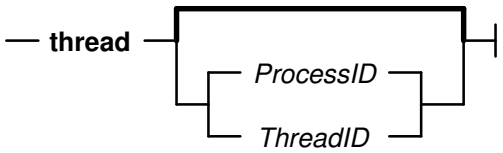
```
swap 1
```

thread Command for the Kernel Debug Program

Purpose

Displays thread table entries.

Syntax



Description

The **thread** command displays the contents of the kernel thread table. If the *ProcessID* parameter is given, information about all kernel threads belonging to that process is displayed. If the *ThreadID* parameter is given, detailed information about the specified kernel thread is displayed. If no parameters are given, information about all kernel threads in the kernel thread table is displayed. Note that the *ProcessID* and *ThreadID* parameters share a common name space: even numbers are always used for process IDs, whereas odd numbers are used for threads (the init processes, PID 1, is an exception).

Examples

1. To display information about all threads in the thread table, enter:

   ```
   thread
   ```

The output is similar to:

```
SLT ST TID   PID CPUID  POLICY PRI CPU  EVENT  PROCNAME      FLAGS
  0 s    3    0   ANY  OTHER  10  78          swapper  0x00001400
  1 s  103    1   ANY  OTHER  3C   0          init     0x00000400
  2*r  205  204     0  OTHER  7F  78          wait     0x00001000
  3 r  307  306     1  OTHER  7F  78          wait     0x00001000
  4 s  409  408   ANY  OTHER  24   0          netm     0x00001000
  5 s  50B  50A   ANY  OTHER  24   0          gil      0x00001000
  6 s  60D  50A   ANY  OTHER  24   0 000B2DA8 gil      0x00001000
  7 s  70F  50A   ANY  OTHER  24   0 000B2DA8 gil      0x00001000
  8 s  811  50A   ANY  OTHER  24   0 000B2DA8 gil      0x00001000
  9 s  913  50A   ANY  OTHER  24   1 000B2DA8 gil      0x00001000
 10 s  A15  60C   ANY  OTHER  3C   0          sh       0x00000400
 11 s  B17  70E   ANY  OTHER  3C   0          sh       0x00000400
```

2. To display information about the threads in process 2106, enter:

```
th 2106
```

3. To display information about the thread with thread ID 1497, enter:

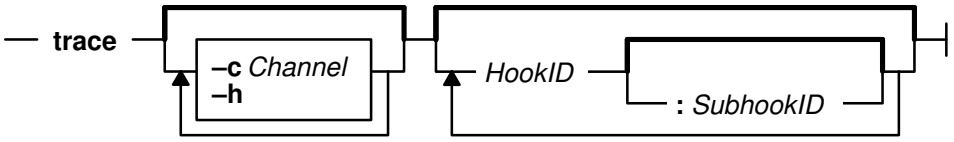
```
th 1497
```

trace Command for the Kernel Debug Program

Purpose

Displays formatted kernel trace buffers.

Syntax



Description

The **trace** command displays the last 128 entries of a kernel trace buffer in reverse chronological order. There are 8 trace buffers, each associated with a trace channel. Each can trace any combination of trace events. Trace data gives an indication of system activity at a very low level; interrupts, input/output, and process scheduling are examples of event types that can be traced.

The **trace** command displays headers for the trace buffers that contain pointers into the trace buffers and the state of the trace driver. Following this are the last 128 entries from the selected trace buffer. Trace entries consist of a major and a minor number for the trace hook, an ASCII trace ID, an ASCII trace hook type, followed by either a hexadecimal dump of the trace data or a pointer to the start of a variable-length block of trace data.

The **trace** command is not meant to replace the **trcfmt** command, which formats the trace data in more detail. It is a facility for viewing system trace data in the event of a system crash before the data has been written to disk.

Flags

–c *Channel* Specifies the trace channel used.

–h Displays the trace headers.

Examples

1. To display a sequence of trace entries, enter:

   ```
   trace
   ```

   The system then returns the following question:

   ```
   Display channel (0 - 8): 0
   ```

2. To display a sequence of trace entries with hookword `105`, enter:

   ```
   trace 105 -c 0
   ```

3. To display a sequence of trace entries with hookword `105` and subhook `d`, enter:

   ```
   trace 105:d -c 0
   ```

4. To display all entries with hookword `105` or `10b`, enter:

   ```
   trace 105 10b
   ```

5. To display all entries with hookword `105` and a `300` in the trace data, enter:

   ```
   trace 105 #300
   ```

6. To display the trace headers, enter:

   ```
   trace -h
   ```

trb Command for the Kernel Debug Program

Purpose

Displays the timer request blocks (TRBs).

Syntax

— **trb** —┤

Description

The **trb** command displays a menu of commands to display timer request block (TRB) information.

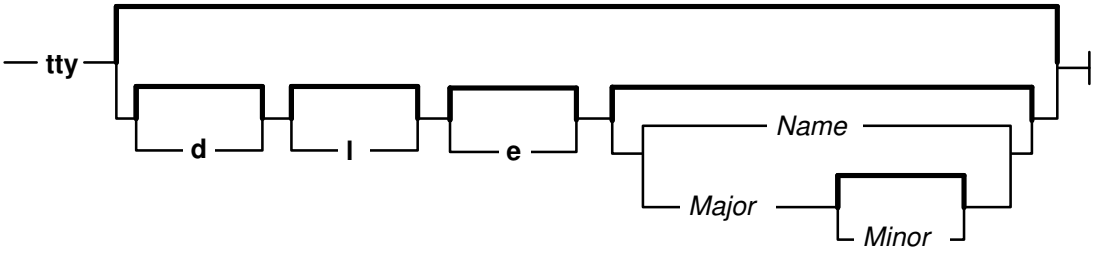
The **trb** command allows you to traverse the active and free TRB chains; examine TRBs by process, slot number, or address; and examine the clock interrupt handler information.

tty Command for the Kernel Debug Program

Purpose

Displays the tty structure.

Syntax



Description

The **tty** command displays tty data structures. If no parameters are specified, a short listing of all opened terminals is displayed. Selected terminals can be displayed by specifying the terminal name in the *Name* parameter, such as **tty1**, or a major device number with optional minor numbers. If the *Major* parameter is specified, all terminals with the specified major number are listed. If the *Major* and *Minor* parameters are both specified, the terminal with both the specified major and minor numbers is listed.

Selected type of information can be displayed, according to the specified flags.

Flags

a	Displays a short listing of all terminals.
o	Displays a short listing of all open terminals.
v	Displays a verbose listing.
d	Displays the driver information.
l	Displays the line discipline information.
e	Displays information for every module and driver present in the stream for the selected lines.

Examples

1. To display listings for each open terminal, enter:

   ```
   tty
   ```

2. To display the driver and line disicipline information for terminal **tty1**, enter:

   ```
   tty d l tty1
   ```

3. To display the listing for the terminal with a major number 7 and a minor number 1, enter:

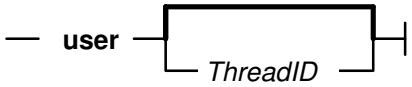
   ```
   tty 7 1
   ```

user Command for the Kernel Debug Program

Purpose

Displays the U-area (user area).

Syntax



Description

The **user** command with no parameter specified displays the U-area for the currently running thread. If the *ThreadID* parameter is specified, then the U-area for that thread is displayed in detail.

Examples

1. To display the current U-area, enter:

   ```
   user
   ```

2. To display the U-area for the thread with thread ID `315`, enter:

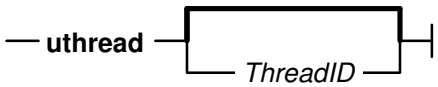
   ```
   u 315
   ```

uthread Command for the Kernel Debug Program

Purpose

Displays the **uthread** structure.

Syntax



Description

The **uthread** command displays **uthread** structures. If the *ThreadID* parameter is given, the **uthread** structure of the specified kernel thread is displayed. Otherwise, the **uthread** structure of the current kernel thread is displayed.

Examples

1. To display the **uthread** structure of the current kernel thread, enter:

   ```
   uthread
   ```

   The output is similar to:

```
using current thread:
UTHREAD AREA FOR TID 0x00000205
SAVED MACHINE STATE
    curid:0x00000204  m/q:0x00000000  iar:0x000214D4  cr:0x24000000
    msr:0x00009030  lr:0x00021504  ctr:0x0002147C  xer:0x20000000
    *prevmst:0x00000000  *stackfix:0x00000000  intpri:0x0000000B
    backtrace:0x00  tid:0x00000000  fpeu:0x00  ecr:0x00000000
    Exception Struct
      0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
    Segment Regs
     0:0x00000000   1:0x007FFFFF   2:0x00000408   3:0x007FFFFF
     4:0x007FFFFF   5:0x007FFFFF   6:0x007FFFFF   7:0x007FFFFF
     8:0x007FFFFF   9:0x007FFFFF  10:0x007FFFFF  11:0x007FFFFF
    12:0x007FFFFF  13:0x007FFFFF  14:0x00000204  15:0x007FFFFF
    General Purpose Regs
```

```
     0:0x00000000   1:0x2FEAEF38   2:0x00270314   3:0x00000054
     4:0x00000002   5:0x00000000   6:0x000BF9B8   7:0x00000000
     8:0xDEADBEEF   9:0xDEADBEEF  10:0xDEADBEEF  11:0x00000000
    12:0x00009030  13:0xDEADBEEF  14:0xDEADBEEF  15:0xDEADBEEF
    16:0xDEADBEEF  17:0xDEADBEEF  18:0xDEADBEEF  19:0xDEADBEEF
    20:0xDEADBEEF  21:0xDEADBEEF  22:0xDEADBEEF  23:0xDEADBEEF
    24:0xDEADBEEF  25:0xDEADBEEF  26:0xDEADBEEF  27:0xDEADBEEF
    28:0xDEADBEEF  29:0xDEADBEEF  30:0xDEADBEEF  31:0xDEADBEEF
    Press "ENTER" to continue, or "x" to exit:>0>
Floating Point Regs
        Fpscr: 0x00000000
     0:0x00000000 0x00000000  1:0x00000000 0x00000000  2:0x00000000 0x00000000
     3:0x00000000 0x00000000  4:0x00000000 0x00000000  5:0x00000000 0x00000000
     6:0x00000000 0x00000000  7:0x00000000 0x00000000  8:0x00000000 0x00000000
     9:0x00000000 0x00000000 10:0x00000000 0x00000000 11:0x00000000 0x00000000
    12:0x00000000 0x00000000 13:0x00000000 0x00000000 14:0x00000000 0x00000000
    15:0x00000000 0x00000000 16:0x00000000 0x00000000 17:0x00000000 0x00000000
    18:0x00000000 0x00000000 19:0x00000000 0x00000000 20:0x00000000 0x00000000
    21:0x00000000 0x00000000 22:0x00000000 0x00000000 23:0x00000000 0x00000000
    24:0x00000000 0x00000000 25:0x00000000 0x00000000 26:0x00000000 0x00000000
    27:0x00000000 0x00000000 28:0x00000000 0x00000000 29:0x00000000 0x00000000
    30:0x00000000 0x00000000 31:0x00000000 0x00000000

Kernel stack address: 0x2FEAEFFC
Press "ENTER" to continue, or "x" to exit:>0>

SYSTEM CALL STATE
    user stack:0x00000000  user msr:0x00000000
    errno address:0xC0C0FADE  error code:0x00  *kjmpbuf:0x00000000
  ut_flags:

PER-THREAD TIMER MANAGEMENT
    Real/Alarm Timer (ut_timer.t_trb[TIMERID_ALRM]) = 0x0
    Virtual Timer (ut_timer.t_trb[TIMERID_VIRTUAL]) = 0x0
    Prof Timer (ut_timer.t_trb[TIMERID_PROF]) = 0x0
    Posix Timer (ut_timer.t_trb[POSIX4]) = 0x0

SIGNAL MANAGEMENT

    *sigsp:0x0  oldmask:hi 0x0,lo 0x0  code:0x0
Press "ENTER" to continue, or "x" to exit:>0>

Miscellaneous fields:
   fstid:0x00000000   ioctlrv:0x00000000   selchn:0x00000000
Uthread area printout terminated.
```

2. To display the uthread structure of the kernel thread with thread ID 1497, enter:

```
ut 1497
```

vars Command for the Kernel Debug Program

Purpose

Displays a list of user-defined variables.

Syntax

— **vars** —|

Description

The **vars** command displays the user-defined variables and their values.

The command displays the variable name and value, and an indication of what is the base of the value. Since the value 10 can be either decimal or hexadecimal it is displayed as `HEX/DEC`. The command displays string variables with no quotes around the string value.

The values of the reserved variables **fx** and **org** are also displayed.

vmm Command for the Kernel Debug Program

Purpose

Displays the virtual memory information menu.

Syntax

— **vmm** —|

Description

The **vmm** command displays a menu of commands for displaying the virtual memory data structures. These commands examine segment register values for kernel segments such as the ram disk and the page space disk maps. Addresses and sizes of VMM data structures are also available, as are VMM statistics such as the number of page faults and the number of pages paged in or out.

xlate Command for the Kernel Debug Program

Purpose

Translates a virtual address to a real address.

Syntax

— **xlate** — *VirtualAddress* —|

Description

The **xlate** command displays the real address corresponding to the specified virtual address.

Example

To display the real address corresponding to the virtual address `10054000`, enter:

```
xlate 10054000
10054000 -virtual- =EF004 -real-
```

`EF004` is the corresponding real address.

Maps and Listings as Tools for the Kernel Debug Program

The assembler listing and the map files are essential tools for debugging using the kernel debugger. In order to create the assembler list file during compilation, use the **–qlist** option while compiling. Also use the **–qsource** option to get the C source listing in the same file:

```
cc -c -DEBUG -D_KERNEL -DIBMR2 demodd.c -qsource -qlist
```

In order to obtain the map file, use the **–bmap:FileName** option on the link editor, enter:

```
ld -o demodd demodd.o -edemoconfig -bimport:/lib/kernex.exp \
-lsys  -lcsys -bmap:demodd.map -bE:demodd.exp
```

You can also create a map file with a slightly different format by using the **nm** command. For example, use the following command to get a map listing for the kernel (/**unix**):

```
nm -xv /unix > unix.m
```

Compiler Listing

The assembler and source listing is used to correlate any C source line with the corresponding assembler lines. The following is a portion of the C source code for a sample device driver. The left column is the line number in the source code:

```
.
.
185
186    if (result = devswadd(devno, &demo_dsw)){
187         printf(”democonfig : failed to add entry points\n”);
188         (void)devswdel(devno);
189         break;
190    }
191    dp->inited = 1;
192    demos_inited++;
193    printf(”democonfig : CFG_INIT success\n”);
194    break;
195
.
.
```

The following is a portion of the assembler listing for the corresponding C code shown previously. The left column is the C source line for the corresponding assembler statement. Each C source line can have multiple assembler source lines. The second column is the offset of the assembler instruction with respect to the kernel extension entry point.

```
.
.
186| 000218 l     80610098  2  L4Z   gr3=devno(gr1,152)
186| 00021C cal   389F0000  1  LR    gr4=gr31
186| 000220 bl    4BFFFDE1  0  CALL  gr3=devswadd,2,
gr3,(struct_4198576)",gr4,devswadd",gr1,cr[01567],gr0",
gr4"-gr12",fp0"-fp13"
186| 000224 cror  4DEF7B82  1
186| 000228 st    9061005C  2  ST4A  #2357(gr1,92)=gr3
186| 00022C st    9061003C  1  ST4A  result(gr1,60)=gr3
186| 000230 l     8061005C  1  L4A   gr3=#2357(gr1,92)
186| 000234 cmpi  2C830000  2  C4    cr1=gr3,0
186| 000238 bc    41860020  3  BT    CL.16,cr1,0x4/eq
187| 00023C ai    307F01A4  1  AI    gr3=gr31,420
187| 000240 bl    4BFFFDC1  2  CALL  gr3=printf,1,'democonfig :
failed to add entry points",gr3,printf",gr1,cr[01567],gr0",
gr4"-gr12",fp0"-fp13"
187| 000244 cror  4DEF7B82  1
188| 000248 l     80610098  2  L4Z   gr3=devno(gr1,152)
188| 00024C bl    4BFFFDB5  0  CALL  gr3=devswdel,1,gr3,
devswdel",gr1,cr[01567],gr0",gr4"-gr12",fp0"-fp13"
188| 000250 cror  4DEF7B82  1
189| 000254 b     48000104  0  B     CL.6
186|                           CL.16:
191| 000258 l     80810040  2  L4Z   gr4=dp(gr1,64)
191| 00025C cal   38600001  1  LI    gr3=1
191| 000260 stb   98640004  1  ST1Z  (char)(gr4,4)=gr3
192| 000264 l     8082000C  1  L4A   gr4=.demos_inited(gr2,0)
192| 000268 l     80640000  2  L4A   gr3=demos_inited(gr4,0)
192| 00026C ai    30630001  2  AI    gr3=gr3,1
192| 000270 st    90640000  1  ST4A  demos_inited(gr4,0)=gr3
193| 000274 ai    307F01D0  1  AI    gr3=gr31,464
193| 000278 bl    4BFFFD89  0  CALL  gr3=printf,1,'democonfig :
CFG_INIT success",gr3,printf",gr1,cr[01567],gr0",gr4"-gr12",
fp0"-fp13"
193| 00027C cror  4DEF7B82  1
194| 000280 b     480000D8  0  B     CL.6
.
.
```

Now with both the assembler listing and the C source listing, you can determine the assembler instruction for a C statement. As an example, consider the C source line at line 191 in the sample code:

```
191             dp->inited = 1;
```

The corresponding assembler instructions are:

```
191| 000258 l     80810040  2  L4Z   gr4=dp(gr1,64)
191| 00025C cal   38600001  1  LI    gr3=1
191| 000260 stb   98640004  1  ST1Z  (char)(gr4,4)=gr3
```

The offsets of these instructions within the sample device driver (demodd) are 000258, 00025C, and 000260.

Map File

The binder map file is a symbol map in address order format. Each symbol listed in the map file has a storage class (CL) and a type (TY) associated with it.

Storage classes correspond to the **XMC_*XX*** variables defined in the **syms.h** file. Each storage class belongs to one of the following section types:

.text Contains read-only data (instructions). Addresses listed in this section use the beginning of the **.text** section as origin. The **.text** section can contain one of the following storage class (CL) values:

- **DB** Debug Table. Identifies a class of sections that has the same characteristics as read only data.
- **GL** Glue Code. Identifies a section that has the same characteristics as a program code. This type of section has code to interface with a routine in another module. Part of the interface code requirement is to maintain TOC addressability across the call.
- **PR** Program Code. Identifies the sections that provide executable instructions for the module.
- **R0** Read Only Data. Identifies the sections that contain constants that are not modified during execution.
- **TB** Reserved.
- **TI** Reserved.
- **XO** Extended Op. Identifies a section of code that is to be treated as a pseudo-machine instruction.

.data Contains read-write initialized data. Addresses listed in this section use the beginning of the **.data** section as origin. The **.data** section can contain one of the following storage class (CL) values:

- **DS** Descriptor. Identifies a function descriptor. This information is used to describe function pointers in languages such as C and Fortran.
- **RW** Read Write Data. Identifies a section that contains data that is known to require change during execution.
- **SV** SVC. Identifies a section of code that is to be treated as a supervisory call.
- **T0** TOC Anchor. Used only by the predefined TOC symbol. Identifies the special symbol TOC. Used only by the TOC header.
- **TC** TOC Entry. Identifies address data that will reside in the TOC.
- **TD** TOC Data Entry. Identifies data that will reside in the TOC.
- **UA** Unclassified. Identifies data that contains data of an unknown storage class.

.bss Contains read-write uninitialized data. Addresses listed in this section use the beginning of the **.data** section as origin. The **.bss** section contain one of the following storage class (CL) values:

- **BS** BSS class. Identifies a section that contains uninitialized data.
- **UC** Unnamed Fortran Common. Identifies a section that contains read write data.

Types correspond to the **XTY**_*XX* variables defined in the **syms.h** file. The type (`TY`) can be one of the following values:

ER	External Reference
LD	Label Definition
SD	Section Definition
CM	BSS Common Definition

The following is a map file for a sample device driver:

```
1  ADDRESS MAP FOR demodd
2                                                    SOURCE-FILE(OBJECT) or
   *IE ADDRESS  LENGTH AL CL TY Sym# NAME       IMPORT-FILE{SHARED-OBJECT}
3  --- -------- ------ -- -- -- ---- ---------- --------------------------
4   I                        ER S1   pinned_heap  /lib/kernex.exp{/unix}
5   I                        ER S2   devswadd     /lib/kernex.exp{/unix}
6   I                        ER S3   devswdel     /lib/kernex.exp{/unix}
7   I                        ER S4   nodev        /lib/kernex.exp{/unix}
8   I                        ER S5   printf       /lib/kernex.exp{/unix}
9   I                        ER S6   uiomove      /lib/kernex.exp{/unix}
10  I                        ER S7   xmalloc      /lib/kernex.exp{/unix}
11  I                        ER S8   xmfree       /lib/kernex.exp{/unix}
12     00000000 0008B8  2 PR SD S9   <>
/tmp/cliff/demodd/demodd.c(demodd.o)
13     00000000           PR LD S10  .democonfig
14     0000039C           PR LD S11  .demoopen
15     000004B4           PR LD S12  .democlose
16     000005D4           PR LD S13  .demoread
17     00000704           PR LD S14  .demowrite
18     00000830           PR LD S15  .get_dp
19     000008B8 000024  2 GL SD S16  <.printf>    glink.s(/usr/lib/glink.o)
20     000008B8           GL LD S17  .printf
21     000008DC 000024  2 GL SD S18  <.xmalloc>   glink.s(/usr/lib/glink.o)
22     000008DC           GL LD S19  .xmalloc
23     00000900 000090  2 PR SD S20  .bzero
noname(/usr/lib/libcsys.a[bzero.o])
24     00000990 000024  2 GL SD S21  <.uiomove>   glink.s(/usr/lib/glink.o)
25     00000990           GL LD S22  .uiomove
26     000009B4 000024  2 GL SD S23  <.devswadd>  glink.s(/usr/lib/glink.o)
27     000009B4           GL LD S24  .devswadd
28     000009D8 000024  2 GL SD S25  <.devswdel>  glink.s(/usr/lib/glink.o)
29     000009D8           GL LD S26  .devswdel
30     000009FC 000024  2 GL SD S27  <.xmfree>    glink.s(/usr/lib/glink.o)
31     000009FC           GL LD S28  .xmfree
32     00000000 000444  4 RW SD S29  <_/tmp/cliff/demodd/demodd$c$>
/tmp/cliff/demodd/demodd.c(demodd.o)
33     00000450 000004  4 RW SD S30  demo_dev
/tmp/cliff/demodd/demodd.c(demodd.o)
34     00000460 000004  4 RW SD S31  demos_inited
/tmp/cliff/demodd/demodd.c(demodd.o)
35     00000470 000080  4 RW SD S32  data
/tmp/cliff/demodd/demodd.c(demodd.o)
36 * E 000004F0 00000C  2 DS SD S33  democonfig
/tmp/cliff/demodd/demodd.c(demodd.o)
37   E 000004FC 00000C  2 DS SD S34  demoopen
/tmp/cliff/demodd/demodd.c(demodd.o)
38   E 00000508 00000C  2 DS SD S35  democlose
/tmp/cliff/demodd/demodd.c(demodd.o)
39   E 00000514 00000C  2 DS SD S36  demoread
/tmp/cliff/demodd/demodd.c(demodd.o)
40   E 00000520 00000C  2 DS SD S37  demowrite
/tmp/cliff/demodd/demodd.c(demodd.o)
41     0000052C 000000  2 T0 SD S38  <TOC>
42     0000052C 000004  2 TC SD S39  <_/tmp/cliff/demodd/demodd$c$>
43     00000530 000004  2 TC SD S40  <printf>
44     00000534 000004  2 TC SD S41  <demo_dev>
45     00000538 000004  2 TC SD S42  <demos_inited>
46     0000053C 000004  2 TC SD S43  <data>
47     00000540 000004  2 TC SD S44  <pinned_heap>
48     00000544 000004  2 TC SD S45  <xmalloc>
49     00000548 000004  2 TC SD S46  <uiomove>
50     0000054C 000004  2 TC SD S47  <devswadd>
51     00000550 000004  2 TC SD S48  <devswdel>
52     00000554 000004  2 TC SD S49  <xmfree>
```

In the sample map file listed previously, the **.data** section starts from the statement at line 32:

```
32     00000000 000444  4 RW SD S29 <_/tmp/cliff/demodd/demodd$c$>
/tmp/cliff/demodd/demodd.c(demodd.o)
```

The TOC (Table of Contents) starts from the statement at line 41:

```
41     0000052C 000000  2 T0 SD S38 <TOC>
```

Using the Kernel Debug Program

This section contains information on setting breakpoints, viewing and modifying global data, displaying registers, and using the stack trace.

Setting Breakpoints

Setting a breakpoint is essential for debugging kernel or kernel extensions. To set a breakpoint, use the following sequence of steps:

1. Locate the assembler instruction corresponding to the C statement.
2. Get the offset of the assembler instruction from the listing.
3. Locate the address where the kernel extension is loaded.
4. Add the address of the assembler instruction to the address where kernel extension is loaded.
5. Set the breakpoint with the **break** command.

The process of locating the assembler instruction and getting its offset is explained in the previous section. The next step is to get the address where the kernel extension is loaded.

Determine the Location of your Kernel Extension

To determine the address where a kernel extension has been loaded, use the following procedure. First, find the load point (the entry point) of the executable kernel extension. This is a label supplied with the **–e** option for the **ld** (links objects) command used while generating the kernel extension. In our example this is the **democonfig** routine.

Then use one of the following six methods to locate the address of this load point. This address is the location where the kernel extension is loaded.

Method 1

If the kernel extension is a device driver, use the **drivers** command to locate the address of the load point routine. The **drivers** command lists all the function descriptors and the function addresses for the device driver (that are in the dev switch table). Usually the **config** routine will be the load point routine. Hence in our example the function address for the **config** (**democonfig**) routine is the address where the kernel extension is loaded.

```
> drivers 255
  MAJ#255      Open         Close        Read         Write
  func   desc  0x01B131B0   0x01B131BC   0x01B131C8   0x01B131D4
  func   addr  0x01B12578   0x01B126A0   0x01B127D4   0x01B12910
               Ioctl        Strategy     Tty          Select
  func   desc  0x00019F10   0x00019F10   0x00000000   0x00019F10
  func   addr  0x00019A20   0x00019A20                0x00019A20
               Config       Print        Dump         Mpx
  func   desc  0x01B131A4   0x00019F10   0x00019F10   0x00019F10
  func   addr  0x01B121EC   0x00019A20   0x00019A20   0x00019A20
               Revoke       Dsdptr       Selptr       Opts
  func   desc  0x00019F10   0x00000000   0x00000000   0x00000002
  func   addr  0x00019A20
```

Method 2

Another method to locate the address is to use the value of the **kmid** pointer returned by the **sysconfig(SYS_KLOAD)** subroutine when loading the kernel extension. The **kmid** pointer points to the address of the load point routine. Hence to get the address of the load point, print the **kmid** value during the **sysconfig** call from the configuration method. Then go into the low level debugger and display the value pointed to by **kmid**. For clarity, set mnemonics for **kmid**.

```
> set kmid 1b131a4
> vars
Listing of the User-defined variables:
  kmid HEX=01B131A4
  fx HEX/DEC=01B1256E
  org
There are 15 free variable slots.
> d kmid
01B131A4   01B121EC 01B131E0 00000000 01B12578
|..!...1.......%x|
> d kmid>
01B121EC   7C0802A6 BFC1FFF8 90010008 9421FF80
||............!..|
```

Method 3

If **kmid** is also not known, use the **find** command to locate the load point routine:

```
> find democonfig 1b00000
01B1256E   66616B65 636F6E66 69677C08 02A693E1
|democonfig|.....|
```

The **find** command will locate the specified string. It initiates a search from the starting address specified in the command. The string that is located is at the end of the **democonfig** routine. Now, backup to locate the beginning of the routine.

Usually all procedures have the instruction 7C0802A6 within the first three or four instructions of the procedure (within the first 12 to 16 bytes). See the assembler listing for the actual position of this instruction within the procedure. Use the **screen** command with the **–** flag to keep going back to locate the instruction. You can help speed up your search by using the ASCII section of the screen output to look for occurrences of the pipe symbol (|), which corresponds to the hexadecimal value 7C, the first byte of the instruction. Once this instruction is found, you can figure out where the start of the procedure is using the assembler listing as a guide.

```
> screen fx
GPR0  000078E4  2FF7FF70  000C5E78  00000000 2FF7FFF8  00000000 00007910 DEADBEEF
GPR8  DEADBEEF  DEADBEEF  DEADBEEF  7C0802A6 DEADBEEF  DEADBEEF DEADBEEF DEADBEEF
GPR16 DEADBEEF  DEADBEEF  DEADBEEF  DEADBEEF DEADBEEF  DEADBEEF DEADBEEF DEADBEEF
GPR24 DEADBEEF  DEADBEEF  DEADBEEF  DEADBEEF DEADBEEF  DEADBEEF DEADBEEF 00007910
MSR   000090B0  CR    00000000  LR  0002506C CTR  000078E4
MQ    00000000  XER   00000000  SRR0    000078E4  SRR1 000090B0 DSISR    40000000
DAR   30000000  IAR   000078E4  (ORG+000078E4)    ORG=00000000  Mode:    VIRTUAL
000078E0    00000000  48000000  4E800020    00000000  |....H...N.. ....|
                       |         b   0x78E4  (000078E4)
000078F0    000C0000  00000000  00000000    00000000  |................|
                                |
01B12560    80020301  00000000  0000036C    000A6661  |...........l..fa|
01B12570    6B65636F  6E666967  7C0802A6    93E1FFFC  |keconfig|.......|
01B12580    90010008  9421FFA0  83E20000    90610078  |.....!.......a.x|
01B12590    9081007C  90A10080  90C10084    307F0294  |...|........0...|
01B125A0    48000535  80410014  80610078    5463043E  |H..5.A...a.xTc.>|
01B125B0    90610038  80610078  48000491    9061003C  |.a.8.a.xH....a.<|
01B125C0    28830000  41860020  8061003C    88630004  |(...A.. .a.<.c..|
```

```
> screen -
.
.
>
>
GPR0   000078E4  2FF7FF70  000C5E78  00000000 2FF7FFF8  00000000 00007910 DEADBEEF
GPR8   DEADBEEF  DEADBEEF  DEADBEEF  7C0802A6 DEADBEEF  DEADBEEF DEADBEEF DEADBEEF
GPR16  DEADBEEF  DEADBEEF  DEADBEEF  DEADBEEF DEADBEEF  DEADBEEF DEADBEEF DEADBEEF
GPR24  DEADBEEF  DEADBEEF  DEADBEEF  DEADBEEF DEADBEEF  DEADBEEF DEADBEEF 00007910
MSR    000090B0  CR     00000000  LR      0002506C  CTR  000078E4 MQ       00000000
XER    00000000  SRR0   000078E4  SRR1    000090B0  DSISR40000000 DAR      30000000
IAR    000078E4  (ORG+000078E4)   ORG=00000000 Mode: VIRTUAL
000078E0     00000000  48000000  4E800020     00000000  |....H...N.. ....|
                        |          b  0x78E4   (000078E4)
000078F0     000C0000  00000000  00000000     00000000  |................|
                         |
01B121E0     00000000  00000000  00000000     7C0802A6  |............|...|
01B121F0     BFC1FFF8  90010008  9421FF80     83E20000  |.........!......|
01B12200     90610098  9081009C  90A100A0     307F0040  |.a..........0..@|
01B12210     80810098  480008C1  80410014     307F0058  |....H....A..0..X|
01B12220     83C20008  63C40000  80A2000C     80C20010  |....c...........|
01B12230     480008A5  80410014  63C30000     80810098  |H....A..c.......|
01B12240     5484043E  90810038  38800000     9081003C  |T..>...88......<|
```

The start of the democonfig routine is at 0x01B121EC.

Method 4

If the load point routine is an exported routine, use the **map** command to locate the appropriate routine:

```
>map <routine name>
```

Method 5

You can also use the **crash** command to locate the load point. After running the **crash** command, run the **le** subcommand to list the load point for all the kernel extensions. The **knlist** subcommand will list the addresses of exported symbols:

```
$ crash
>le
>quit
```

The **le** subcommand shows the module start address. The first procedure in the kernel extension would follow the module header from the module start address. Hence in the case of the example demodd kernel extension, **le** showed the module start address to be 0x01B12000 and the democonfig procedure starts at 0x01B121EC.

You can locate the start of the democonfig procedure by searching for the first instruction of the democonfig procedure which would be usually 0x7C0802A6. Use the assembler listing to determine the first instruction.

First, display memory at 0x01b12000 and then use the **screen** subcommand to search ahead.

```
>screen 01b12000
>screen +
.
.
```

Method 6

Use the **find** command to search for a pattern:

```
> find democonfig 1b00000
01B1256E   66616B65 636F6E66 69677C08 02A693E1
|democonfig|.....|
```

We know that the module starts before 1B1256E. We also know that the “magic” number is 01DF. The loader identifies a file as a load module by looking for 01DF as the first two bytes in the file. So, the greatest address which is less than 1B1256E that contains 01DF, will be the start of the module, provided that it is on a page boundary. This means it has a mask of FFFFF000, a 4096 boundary or 0x1000:

```
> find 01df 01900000 * 2
```

Search starting at 1900000 through the kernel storage (the *) for 01DF on a 2-byte boundary.

The greatest address, on a page boundary, that is less than 1B1256E will be the module start. This will be offset 00000000 in the map file.

Change the Origin

Set the origin to the address of the load point. By default this is zero. By changing the origin to the address of the load point, you can directly correlate the address in the assembler listing with the address for the Instruction Address Register (IAR) and break points.

```
>set fkcfg 1B121EC      set a variable called fkcfg

>origin fkcfg
```

Set the Break Point

Now set the break point with the **break** command. Assume that we want to set the breakpoint at the assembler instruction at offset 218 (using the assembler listing):

```
>break +218             If origin has been set to load point
```

OR

```
>break 1B121EC+218
```

Viewing and Modifying Global Data

You can access the global data with two different methods. To understand how to locate the address of a global variable, we use the example of our demodd device driver. Here we try to view and modify the value of the data[ ] character array in the sample demodd device driver.

Use the first method only when you break in a procedure for the kernel extension to be debugged. You can use the second method at any time.

Method 1

1. After getting into the low level debugger, set a break point at the **demoread** procedure call. You can use any routine in demodd for this purpose.
2. Call the **demoread** routine. When the system breaks in **demoread** and invokes the debugger, the GPR2 (general purpose register 2) points to the TOC address. Now use the offset of the address of any global variable (from the start of TOC) to determine its address. The TOC is listed in the map file.

   The map file on page 15-67 shows that the address of the data[ ] array is at 0x53C while the TOC is at 0x52C. The offset of the address of the data[ ] array with respect to the start of TOC is 0x53C – 0x52C = 0x10. Hence the address of the data[ ] variable is at (r2+10). And the actual data[ ] variable is located at the address value in (r2 + 10):

   ```
   > d r2
   01B131E0 01B12CCC 0004E7D0 01B13114 01B1311C |..,.......1...1.|
   > d r2+10>
   01B13124 61626364 65666768 696A6B6C 6D6E6F70 |abcdefghijklmnop|
   ```

   Now we can change the value of the data[] variable. As an example, we change the first four bytes of data[ ] to “pppp” (p = 70):

   ```
   > st r2+10> 70707070
   > d r2+10>
   01B13124 70707070 65666768 696A6B6C 6D6E6F70 |ppppefghijklmnop|
   ```

Method 2

You can use this method at any time. This method requires the map file and the address at which the relevant kernel address has been loaded. This method currently works because of the manner in which a kernel extension is loaded. But it may not work if the procedure for loading a kernel extension changes.

The address of a variable is:

Address of the last function before the variable in the map file	+	Length of the function	+	Offset of the variable

The following is the section of the map file (see page 15-67) showing the data[ ] variable and the last function (xmfree) in the **.text** section:

```
26     000009B4 000024  2 GL SD S23  <.devswadd>  glink.s(/usr/lib/glink.o)
27     000009B4           GL LD S24  .devswadd
28     000009D8 000024  2 GL SD S25  <.devswdel>  glink.s(/usr/lib/glink.o)
29     000009D8           GL LD S26  .devswdel
30     000009FC 000024  2 GL SD S27  <.xmfree>    glink.s(/usr/lib/glink.o)
31     000009FC           GL LD S28  .xmfree
32     00000000 000444  4 RW SD S29  <_/tmp/cliff/demodd/demodd$c$>
/tmp/cliff/demodd/demodd.c(demodd.o)
33     00000450 000004  4 RW SD S30  demo_dev
/tmp/cliff/demodd/demodd.c(demodd.o)
34     00000460 000004  4 RW SD S31  demos_inited
/tmp/cliff/demodd/demodd.c(demodd.o)
35     00000470 000080  4 RW SD S32  data
/tmp/cliff/demodd/demodd.c(demodd.o)
36 * E 000004F0 00000C  2 DS SD S33  democonfig
/tmp/cliff/demodd/demodd.c(demodd.o)
37   E 000004FC 00000C  2 DS SD S34  demoopen
/tmp/cliff/demodd/demodd.c(demodd.o)
```

The last function in the **.text** section is at lines 30–31. The offset address of this function from the map is 0x000009FC (line 30, column 2). The length of the function is 0x000024 (line 30, column 3). The offset address of the data[ ] variable is 0x000000470 (line 35, column 2). Hence the offset of the address of the data[ ] variable is:

```
0x000009FC + 0x000024 + 0x00000470 = 0x00000E90
```

Add this address value to the load point value of the demodd kernel extension. If, as in the case of the sample demodd device handler, this is 0x1B131A4, then the address of the data[ ] variable is:

```
0x1B121EC + 0x00000E90 = 0x1B1307C

>display 1B1307C
01B1307C  61626364 65666768 696A6B6C 6D6E6F70  |abcdefghijklmnop|
```

Now change the value of the data[ ] variable as in method 1.

Note that in method 1, using the TOC, you found the address of the address of data[ ], while in method 2 you simply found the address of data[ ].

Displaying Registers on a Micro Channel Adapter

When you write a device driver for a new Micro Channel adapter, you often want to be able to read and write to registers that reside on the adapter. This is a way of seeing if the hardware is functioning correctly. For example, to examine a register on the Token Ring adapter, first see where this adapter resides in the bus I/O space:

```
$lsdev -C

sys0        Available 00-00 System Object
sysunit0    Available 00-00 System Unit
sysplanar0  Available 00-00 CPU Planar
.
.
scsi0       Available 00-01 SCSI I/O Controller
tok0        Available 00-02 Token-Ring High-Performance Adapter
ent0        Available 00-03 Ethernet High-Performance LAN Adapter


$lsattr -l tok0 -E

bus_intr_lvl          3     Bus interrupt level False
intr_priority         3     Interrupt priority  False
.
.
rdto        92        RECEIVE DATA TRANSFER OFFSET      True
bus_io_addr 0x86a0    Bus I/O address                   False
dma_lvl     0x5       DMA arbitration level             False
dma_bus_mem 0x202000  Address of bus memory used DMA    False
```

We now know that the token ring adapter is located at 0x86A0.

To read a specific register, enter the kernel debugger and use the **sregs** command to display the segment registers. Find an unused segment register (=007FFFFF). For this example, assume s9 is not used. Enable the Micro Channel bus addressing with the **set** command:

```
set s9 820c0020
```

Use the **sregs** command to display the segment register values to check that you typed it in correctly.

From the *POWERstation and POWERserver Hardware Technical Information-Options and Devices*, we know that the address of the Adapter Communication and Status register is P6a6. The value of P is based on the Bus I/O address (bus_io_addr) of the adapter. In the above example, this is 86A0. It could have been anything from 86A0 to F6A0 on a 0x1000 byte boundary. Hence P is 8, and the address of the Communication and Status register is 86A6. The **display** command now displays the two-byte register:

```
d 900086a6 2
```

The key is to load a segment register with 820c0020 and then use that segment register to reference registers and memory on your adapter. You can use the same method to access registers resident on the IOCC. In that case, load the segment register with a value of 820c00e0.

Stack Trace

The stack trace gives the stack history which provides the sequence of procedure calls leading to the current IAR. The **Ret Addr** is the address of the instruction calling this procedure. You can use the map file to locate the name of the procedure. Note that the first stack frame shown is almost always useless, since data either has not been saved yet, or is from a previous call. The last function preceding the **Ret Addr** is the function that called the procedure.

You can also use the **map** command to locate the function name if the function was exported. The **map <addr>** command locates the symbol before the given address. The following is a concise view of the stack:

```
Low                     |                     |Stack grows at
Addresses               |                     |this end.
                        |---------------------|
Callee's stack -> 0     |     Back chain      |
pointer           4     |      Saved CR       |
                  8     |      Saved LR       |
              12-16     |      Reserved       |<---LINK AREA (callee)
                 20     |      SAVED TOC      |
                        |---------------------|
Space for P1-P8         |         P1          |  OUTPUT ARGUMENT AREA
is always reserved      |        ...          |<---(Used by callee to
                        |         Pn          |    construct argument
                        |      Callee's       |
                        |       stack         | <--- LOCAL STACK AREA
                        |        area         |
                        |---------------------|
                        |                     | (Possible word wasted
                        |---------------------|    for alignment.)
-8*nfprs-4*ngprs -->    |   Caller's GPR      | Rfirst = R13 for full
save                    |     save area       |          save
                        |    max 19 words     | R31
                        |---------------------|
-8*nfprs -->            |   Caller's FPR      | Ffirst = F14 for a
                        |     save area       |          full save
                        |   max 18 dblwds     | F31
                        |---------------------|
Caller's stack -> 0     |     Back chain      |
pointer           4     |      Saved CR       |
                  8     |      Saved LR       |
              12-16     |      Reserved       |<---LINK AREA (caller)
                 20     |      Saved TOC      |
                        |---------------------|
Space for P1-P8   24    |         P1          |  INPUT PARAMETER AREA
is always reserved      |        ...          | <---(Callee's input
                        |         Pn          |     parameters found
                        |---------------------|      here. Is also
                        |      Caller's       |   caller's arg area.)
                        |       stack         |
High                    |        area         |
Addresses               |
```

The following is a sample stack history with a break in the sample **demodd** kernel extension. The breakpoint was set at the start of the **demoread** routine at 0x1B127D4 (Beginning IAR). This was called from an instruction at 0x000824B0 (**Ret Addr**). This in turn is called by the instruction at address 0x00085F54 (**Ret Addr**), and so on.

The low values of the addresses (0x000824B0 and 0x00085F54) suggest that the instructions are in /**unix**. You can use the **crash** command and the **le** subcommand to determine the right kernel extension that is loaded in an address range.

0x1b127d4 beginning **demoread** in **demodd**
0x000824b0 **.rdevread** in /**unix**
0x00085f54 **.cdev_rdwr** in /**unix**

```
> stack
Beginning IAR: 0x01B127D4      Beginning Stack: 0x2FF97C28
Chain:0x2FF97C88  CR:0x24222082  Ret Addr:0x000824B0  TOC:0x000C5E78
P1:0x2003F800  P2:0x2003F800  P3:0x0000008C  P4:0x00000001
P5:0x01B11200  P6:0x00000000  P7:0x2FF97D38  P8:0x00000000
2FF97C60   00000203 00000000 2FF97CF8 2FF7FCD0   |......../.|./...|
2FF97C70   29057E6B 00001000 2FF97DC0 018E8BE0   |).~k..../.}.....|
2FF97C80   00FF0000 00000000 2FF97CD8 22222044   |......../.|."" D|
Returning to Stack frame at 0x2FF97C88
Press ENTER to continue or x to exit:
```

```
>

Chain:0x2FF97CD8  CR:0x22222044  Ret Addr:0x00085F54  TOC:0x00000000
P1:0x00000000  P2:0x018C41E0  P3:0x2FF97CF8  P4:0x2FF7FCC8
P5:0x000850E0  P6:0x00000000  P7:0xDEADBEEF  P8:0xDEADBEEF
2FF97CC0   DEADBEEF DEADBEEF 00000000 000BE4F8   |................|
2FF97CD0   001E70F8 000BE7A4 2FF97D28 000BE5AC   |..p...../.}(....|
Returning to Stack frame at 0x2FF97CD8
Press ENTER to continue or x to exit:
...
>
Chain:0x00000000  CR:0x22222022  Ret Addr:0x0000238C  TOC:0x00000000
P1:0x00000003  P2:0x30000000  P3:0x00000800  P4:0x00000000
P5:0x00000000  P6:0x00000000  P7:0x00000000  P8:0x00000000
Returning to Stack frame at 0x0
Press ENTER to continue or x to exit:
> Trace back complete.
```

Error Messages for the Kernel Debug Program

The following error messages can appear while using the kernel debug program:

1. `Bad type -- trace terminated.`

   A trace event was found that had an incorrect hookword type, and the traceback was terminated. This message is for your information only.

2. `Channel out of range.`

   You entered a value that is outside of the numeric range of acceptable channel numbers. Enter the command again, selecting a channel in the range displayed in the prompt.

3. `Do you want to continue the search? (Y/N)`

   Ten consecutive pages were not in storage. To continue the search, enter `Y` (yes). To exit the search enter `N` (no).

4. `The address you specified is not in real storage.`

   The command was rejected because the data at the address you specified has been paged out of RAM to disk. Enter the command again with a data address that is currently in RAM.

5. `The page at` *`Address`* `is not in real storage.`

   The search passed over a page that was not in storage. Action is not required. This message is for your information only.

6. `The value cannot be found.`

   You specified a value that cannot be found or was not in real storage. Action is not required. This message is for your information only.

7. `This breakpoint is undefined or not currently addressable.`

   The breakpoint was not cleared because it is undefined or its segment is not currently addressable. Try to load the segment ID into a segment register with the **set** command.

8. `Timestamp paged out.`

   A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.

9. `Trace data paged out.`

   A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.

10. `Trace entry paged out.`

    A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.

11. `Trace header paged out.`

    A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.

12. `Trace Queue header paged out.`

    A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.

13. `You cannot set more than 32 breakpoints.`

    The breakpoint is not set because you tried to set more than the maximum number of breakpoints allowed on the system. Clear at least one breakpoint before setting another breakpoint.

14. `You cannot Step or Go into paged-out storage.`

    The command cannot run because you specified an address for the command that is in paged-out storage. Specify an address that is not in paged-out storage.

15. `You did not enter all required parameters.`

    The command was unsuccessful because you did not specify all the required parameters. Enter the command again with the necessary parameters.

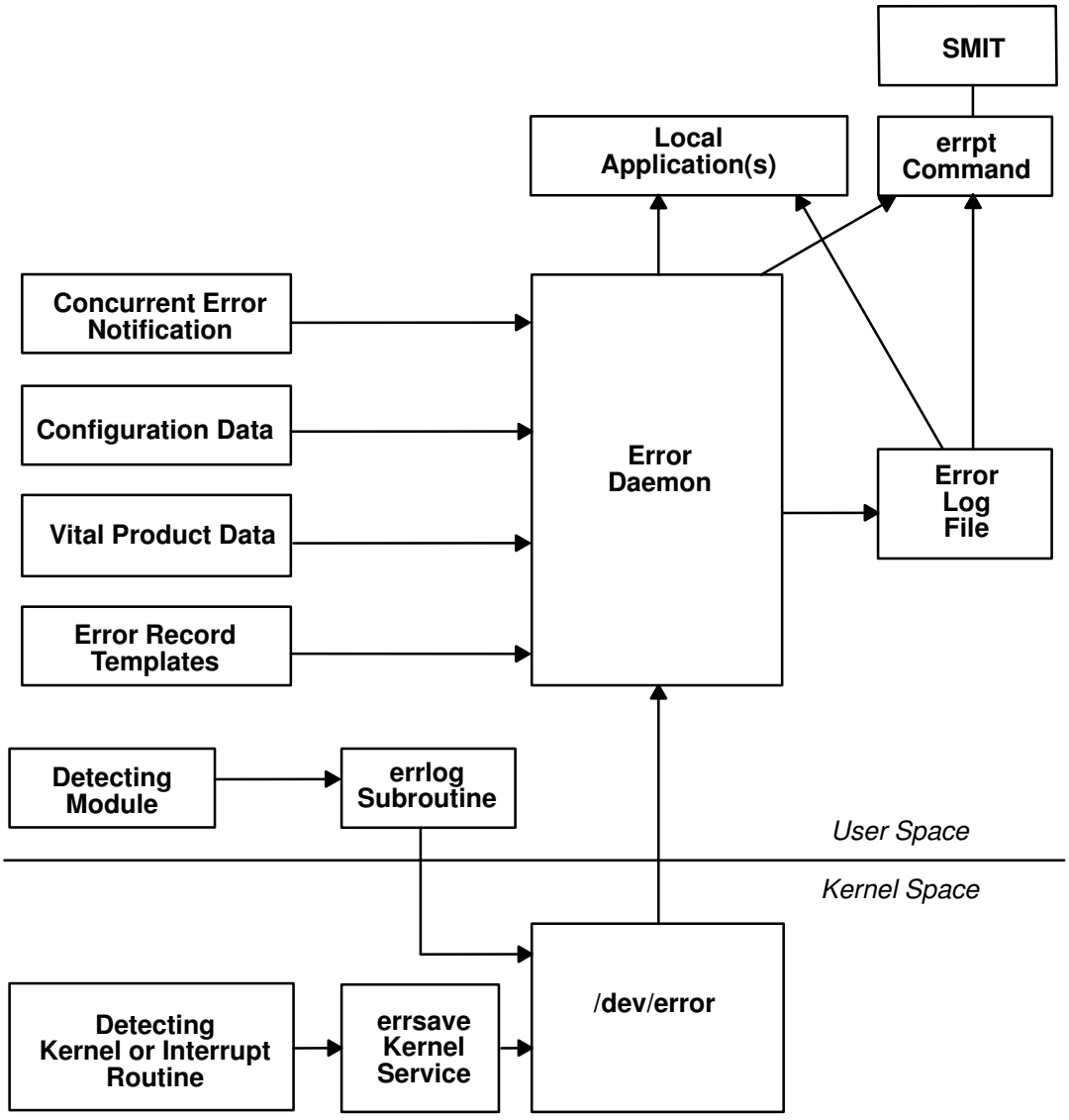
16. `You entered a parameter that is not valid.`

    The command was unsuccessful because you specified a parameter that the debug program did not recognize. Check the spelling and syntax of the parameter you specified. Then, enter the command again with a valid parameter.

Error Logging

The error facility allows a device driver to have entries recorded in the system error log. These error log entries record any software or hardware failures that need to be available either for informational purposes or for fault detection and corrective action. The device driver, using the **errsave** kernel service, adds error records to the special file /**dev**/**error**.

The **errdemon** daemon then picks up the error record and creates an error log entry. When you access the error log either through SMIT (System Management Interface Tool) or with the **errpt** command, the error record is formatted according to the error template in the error template repository and presented in either a summary or detailed report. See the Flow of the Error Logging Facility figure on page 15-78 for an illustration of this.



Flow of the Error Logging Facility

Precoding Steps to Consider

Follow three precoding steps before initiating the error logging process. It is beneficial to understand what services are available to developers, and what the customer, service personnel, and defect personnel see.

Determine the Importance of the Error

The first preceding step is to review the error-logging documentation and determine whether a particular error should be logged. Do not use system resources for logging information that is unimportant or confusing to the intended audience.

It is, however, a worse mistake *not* to log an error that merits logging. You should work in concert with the hardware developer, if possible, to identify detectable errors and the information that should be relayed concerning those errors.

Determine the Text of the Message

The next step is to determine the text of the message. Use the **errmsg** command with the **–w** flag to browse the system error messages file for a list of available messages. If you are developing a product for wide-spread general distribution and do not find a suitable system error message, you can submit a request to your supplier for a new message or follow the procedures that your organization uses to request new error messages. If your product is an in-house application, you can use the **errmsg** command to define a new message that meets your requirements.

Determine the Correct Level of Thresholding

Finally, determine the correct level of thresholding. Each error to be logged, regardless of whether it is a software or hardware error, can be limited by thresholding to avoid filling the error log with duplicate information.

Side effects of runaway error logging include overwriting existing error log entries and unduly alarming the end user. The error log is not unlimited in size. When its size limit is reached, the log wraps. If a particular error is repeated needlessly, existing information is overwritten, possibly causing inaccurate diagnostic analyses. The end user or service person can perceive a situation as more serious or pervasive than it is if they see hundreds of identical or nearly identical error entries.

You are responsible for implementing the proper level of thresholding in the device driver code.

The error log currently equals 1MB. As shipped, it cleans up any entries older than 30 days. In order to ensure that your error log entries are actually informative, noticed, and remain intact, *test your driver thoroughly*.

Coding Steps

To begin error logging,

1. Select the error text.
2. Construct error record templates.
3. Add error logging calls into the device driver code.

Selecting the Error Text

The first task is to select the error text. After browsing the contents of the system message file, three possible paths exist for selecting the error text. Either all of the desired messages for the new errors exist in the message file, none of the messages exist, or a combination of errors exists.

- If the messages required already exist in the system message file, make a note of the four-digit hexadecimal identification number, as well as the message-set identification letter. For instance, a desired error description can be:

```
SET E
E859 ”The wagon wheel is broken.”
```

- If none of the system error messages meet your requirements, and if you are responsible for developing a product for wide-spread general distribution, you can either contact your supplier to allocate new messages or follow the procedures that your organization uses to request new messages. If you are creating an in-house product, use the **errmsg**

command to write suitable error messages and use the **errinstall** command to install them. Refer to “Software Product Packaging” in *AIX Version 4.1 General Programming Concepts: Writing and Debugging Programs* for more information. Take care not to overwrite other error messages.

- It is also possible to use a combination of existing messages and new messages within the same error record template definition.

Constructing Error Record Templates

The second step is to construct your *error record templates*. An error record template defines the text that appears in the error report. Each error record template has the following general form:

```
Error Record Template
        +LABEL:
                Comment =
                Class =
                Log =
                Report =
                Alert =
                Err_Type =
                Err_Desc =
                Probable_Causes =
                User_Causes =
                User_Actions =
                Inst_Causes =
                Inst_Actions =
                Fail_Causes =
                Fail_Actions =
                Detail_Data = <data_len>, <data_id>, <data_encodi
ng>
```

Each field in this stanza has well-defined criteria for input values. See the **errupdate** command for more information. The fields are:

`Label` Requires a unique label for each entry to be added. The label must follow C language rules for identifiers and must not exceed 16 characters in length.

`Comment` Indicates this is a comment field. You must enclose the comment in double quotation marks; and it cannot exceed 40 characters.

`Class` Requires class values of **H** (hardware), **S** (software), or **U** (Undetermined).

`Log` Requires values True or False. If failure occurs, the errors are logged only if this field value is set to True. When this value is False the `Report` and `Alert` fields are ignored.

`Report` The values for this field are True or False. If the logged error is to be displayed using error report, the value of this field must be True.

`Alert` Set this field to True for errors that are alertable. For errors that are not alertable, set this field to False.

`Err_Type` Describes the severity of the failure that occurred. Possible values are INFO, PEND, PERF, PERM, TEMP, and UNKN where:

INFO The error log entry is informational and was not the result of an error.

PEND A condition in which it is determined that the loss of availability of a device or component is imminent.

PERF A condition in which the performance of a device or component was degraded below an acceptable level.

PERM A permanent failure is defined as a condition that was not recoverable. For example, an operation was retried a prescribed number of times without success.

TEMP Recovery from this temporary failure was successful, yet the number of unsuccessful recovery attempts exceeded a predetermined threshold.

UNKN A condition in which it is not possible to assess the severity of a failure.

`Err_Desc` Describes the failure that occurred. Proper input for this field is the four-digit hexadecimal identifier of the error description message to be displayed from `SET E` in the message file.

`Prob_Causes` Describes one or more probable causes for the failure that occurred. You can specify a list of up to four `Prob_Causes` identifiers separated by commas. A `Prob_Causes` identifier displays a probable cause text message from `SET P` in the message file. List probable causes in the order of decreasing probability. At least one probable cause identifier is required.

`User_Causes` Specifies a condition that an operator can resolve without contacting any service organization. You can specify a list of up to four `User_Causes` identifiers separated by commas. A `User_Causes` identifier displays a text message from `SET U` in the message file. List user causes in the order of decreasing probability. Leave this field blank if it does not apply to the failure that occurred. If this field is blank, either the `Inst_Causes` or the `Fail_Causes` field must not be blank.

`User_Actions`
Describes recommended actions for correcting a failure that resulted from a user cause. You can specify a list of up to four recommended `User_Actions` identifiers separated by commas. A recommended `User_Actions` identifier displays a recommended action text message, `SET R` in the message file. You must leave this field blank if the `User_Causes` field is blank.

The order in which the recommended actions are listed is determined by the expense of the action and the probability that the action corrects the failure. Actions that have little or no cost and little or no impact on system operation should always be listed first. When actions for which the probability of correcting the failure is equal or nearly equal, list the least expensive action first. List remaining actions in order of decreasing probability.

`Inst_Causes` Describes a condition that resulted from the initial installation or setup of a resource. You can specify a list of up to four `Inst_Causes` identifiers separated by commas. An `Inst_Causes` identifier displays a text message, `SET I` in the message file. List the install causes in the order of decreasing probability. Leave this field blank if it is not applicable to the failure that occurred. If this field is blank, either the `User_Causes` or the `Failure_Causes` field must not be blank.

`Inst_Actions`
Describes recommended actions for correcting a failure that resulted from an install cause. You can specify a list of up to four recommended `Inst_actions` identifiers separated by commas. A recommended `Inst_actions` identifier identifies a recommended action text message, `SET R` in the message file. Leave this field blank if the `Inst_Causes` field is blank. The order in which the recommended actions are listed is determined by the expense of the action and the probability that the action corrects the failure. See the `User_Actions` field for the list criteria.

`Fail_Causes` Describes a condition that resulted from the failure of a resource. You can specify a list of up to four `Fail_Causes` identifiers separated by commas. A `Fail_Causes` identifier displays a failure cause text message, `SET F` in the message file. List the failure causes in the order of decreasing probability. Leave this field blank if it is not applicable to the failure that occurred. If you leave this field blank, either the `User_Causes` or the `Inst_Causes` field must not be blank.

`Fail_Actions`
Describes recommended actions for correcting a failure that resulted from a failure cause. You can specify a list of up to four recommended action identifiers separated by commas. The `Fail_Actions` identifiers must correspond to recommended action messages found in `SET R` of the message file. Leave this field blank if the `Fail_Causes` field is blank. Refer to the description of the `User_Actions` field for criteria in listing these recommended actions.

`Detail_Data` Describes the detailed data that is logged with the error when the failure occurs. The `Detail_data` field includes the name of the detecting module, sense data, or return codes. Leave this field blank if no detailed data is logged with the error.

You can repeat the `Detail_Data` field. The amount of data logged with an error must not exceed the maximum error record length defined in the **sys/err_rec.h** header file. Save failure data that cannot be contained in an error log entry elsewhere, for example in a file. The detailed data in the error log entry contains information that can be used to correlate the failure data to the error log entry. Three values are required for each detail data entry:

data_len Indicates the number of bytes of data to be associated with the **data_id** value. The **data_len** value is interpreted as a decimal value.

data_id Identifies a text message to be printed in the error report in front of the detailed data. These identifiers refer to messages in `SET D` of the message file.

data_encoding
Describes how the detailed data is to be printed in the error report. Valid values for this field are:

ALPHA The detailed data is a printable ASCII character string.

DEC The detailed data is the binary representation of an integer value, the decimal equivalent is to be printed.

HEX The detailed data is to be printed in hexadecimal.

Sample Error Record Template

An example of an error record template is:

```
+ MISC_ERR:
        Comment = "Interrupt: I/O bus timeout or channel check"
        Class = H
        Log = TRUE
        Report = TRUE
        Alert = FALSE
        Err_Type = UNKN
        Err_Desc = E856
        Prob_Causes = 3300, 6300
        User_Causes =
        User_Actions =
        Inst_Causes =
        Inst_Actions =
        Fail_Causes = 3300, 6300
        Fail_Actions = 0000
        Detail_Data = 4, 8119, HEX      *IOCC bus number
        Detail_Data = 4, 811A, HEX      *Bus Status Register
        Detail_Data = 4, 811B, HEX      *Misc. Interrupt Register
```

Construct the error templates for all new errors to be added in a file suitable for entry with the **errupdate** command. Run the **errupdate** command with the **–h** flag and the input file. The new errors are now part of the error record template repository. A new header file is also created (**file.h**) in the same directory in which the **errupdate** command was run. This header file must be included in the device driver code at compile time. Note that the **errupdate** command has a built-in syntax checker for the new stanza that can be called with the **–c** flag.

Adding Error Logging Calls into the Code

The third step in coding error logging is to put the error logging calls into the device driver code. The **errsave** kernel service allows the kernel and kernel extensions to write to the error log. Typically, you define a routine in the device driver that can be called by other device driver routines when a loggable error is encountered. This function takes the data passed to it, puts it into the proper structure and calls the **errsave** kernel service. The syntax for the **errsave** kernel service is:

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

void errsave(buf, cnt)
char *buf;
unsigned int cnt;
```

where,

buf Specifies a pointer to a buffer that contains an error record as described in the **sys/errids.h** header file.

cnt Specifies a number of bytes in the error record contained in the buffer pointed to by the *buf* parameter.

The following sample code is an example of a device driver error logging routine. This routine takes data passed to it from some part of the main body of the device driver. This code simply fills in the structure with the pertinent information, then passes it on using the **errsave** kernel service.

```
void
errsv_ex (int err_id, unsigned int port_num,
            int line, char *file, uint data1, uint data2)
{
    dderr   log;
    char      errbuf[255];
    ddex_dds  *p_dds;

    p_dds = dds_dir[port_num];
    log.err.error_id = err_id;

    if (port_num = BAD_STATE) {
            sprintf(log.err.resource_name, "%s :%d",
              p_dds->dds_vpd.adpt_name, data1);
            data1 = 0;
    }


else
                sprintf(log.err.resource_name,"%s",p_dds->dds_vpd
.devname);

        sprintf(errbuf, "line: %d file: %s", line, file);
        strncpy(log.file, errbuf, (size_t)sizeof(log.file));

        log.data1 = data1;
        log.data2 = data2;

        errsave(&log, (uint)sizeof(dderr));   /* run actual loggi
ng  */
}  /* end errlog_ex */
```

The data to be passed to the **errsave** kernel service is defined in the **dderr** structure which is defined in a local header file, **dderr.h**. The definition for **dderr** is:

```
typedef struct dderr {
        struct  err_rec0 err;
        int  data1;     /* use data1 and data2 to show detail  */
        int  data2;     /* data in the errlog report. Define   */
                        /* these fields in the errlog template */
                        /* These fields may not be used in all */
                        /* cases.                              */
} dderr;
```

The first field of the **dderr.h** header file is comprised of the **err_rec0** structure, which is defined in the **sys/err_rec.h** header file. This structure contains the ID (or label) and a field for the resource name. The two data fields hold the detail data for the error log report. As an alternative, you could simply list the fields within the function.

You can also log a message into the error log from the command line. To do this, use the **errlogger** command.

After you add the templates using the **errupdate** command, compile the device driver code along with the new header file. Simulate the error and verify that it was written to the error log correctly. Some details to check for include:

- Is the error demon running? This can be verified by running the **ps –ef** command and checking for `/usr/lib/errdemon` as part of the output.
- Is the error part of the error template repository? Verify this by running the **errpt –at** command.

- Was the new header file, which was created by the **errupdate** command and which contains the error label and unique error identification number, included in the device driver code when it was compiled?

Writing to the /dev/error Special File

The error logging process begins when a loggable error is encountered and the device driver error logging subroutine sends the error information to the **errsave** kernel service. The error entry is written to the /**dev**/**error** special file. Once the information arrives at this file, it is time-stamped by the **errdemon** daemon and put in a buffer. The **errdemon** daemon constantly checks the /**dev**/**error** special file for new entries, and when new data is written, the daemon collects other information pertaining to the resource reporting the error. The **errdemon** daemon then creates an entry in the /**var**/**adm**/**ras**/**errlog** error logging file.

Performance Tracing

The AIX **trace** facility is useful for observing a running device driver and system. The **trace** facility captures a sequential flow of time-stamped system events, providing a fine level of detail on system activity. Events are shown in time sequence and in the context of other events. The **trace** facility is useful in expanding the trace event information to understand who, when, how, and even why the event happened.

Introduction

The operating system is shipped with permanent trace event points. These events provide general visibility to system execution. You can extend the visibility into applications by inserting additional events and providing formatting rules.

Care was taken in the design and implementation of this facility to make the collection of **trace** data efficient, so that system performance and flow would be minimally altered by activating **trace**. Because of this, the facility is extremely useful as a performance analysis tool and as a problem determination tool.

The **trace** facility is more flexible than traditional system monitor services that access and present statistics maintained by the system. With traditional monitor services, data reduction (conversion of system events to statistics) is largely coupled to the system instrumentation. For example, the system can maintain the minimum, maximum, and average elapsed time observed for runs of a task and permit this information to be extracted.

The **trace** facility does not strongly couple data reduction to instrumentation, but provides a stream of system events. It is not required to presuppose what statistics are needed. The statistics or data reduction are to a large degree separated from the instrumentation.

You can choose to develop the minimum, maximum, and average time for task A from the flow of events. But it is also possible to extract the average time for task A when called by process B, extract the average time for task A when conditions XYZ are met, develop a standard deviation for task A, or even decide that some other task, recognized by a stream of events, is more meaningful to summarize. This flexibility is invaluable for diagnosing performance or functional problems.

The **trace** facility generates large volumes of data. This data cannot be captured for extended periods of time without overflowing the storage device. This allows two practical ways that the **trace** facility can be used natively.

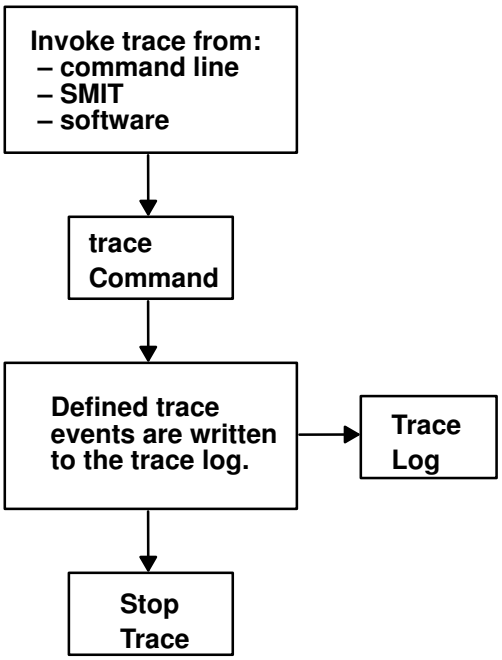
First, the **trace** facility can be triggered in multiple ways to capture small increments of system activity. It is practical to capture seconds to minutes of system activity in this way for post-processing. This is sufficient time to characterize major application transactions or interesting sections of a long task.

Second, the **trace** facility can be configured to direct the event stream to standard output. This allows a realtime process to connect to the event stream and provide data reduction in real-time, thereby creating a long term monitoring capability. A logical extension for specialized instrumentation is to direct the data stream to an auxiliary device that can either store massive amounts of data or provide dynamic data reduction.

You can start the system trace from:

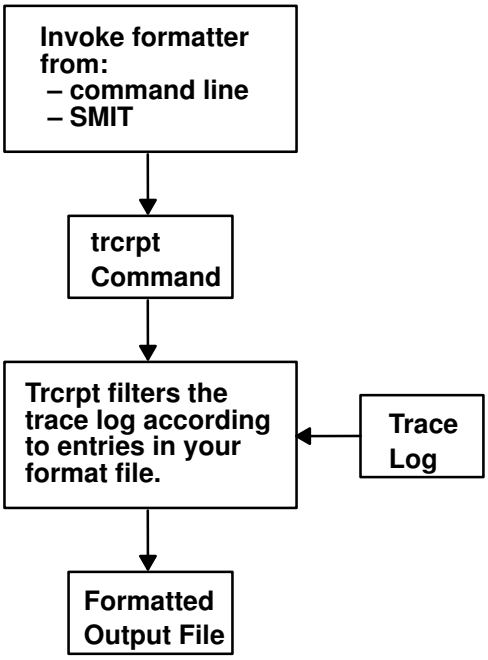
- The command line
- SMIT
- Software

As shown in the following Starting and Stopping Trace figure, the trace facility causes predefined events to be written to a trace log. The tracing action is then stopped. Tracing from a command line is discussed in “Controlling Trace” on page 15-90. Tracing from a software application is discussed and an example is presented in “Examples of Coding Events and Formatting Events” on page 15-108.



Starting and Stopping Trace

After a trace is started and stopped, you must format it before viewing it. This is illustrated in the following Trace Formatting figure. To format the trace events that you have defined, you must provide a stanza that describes how the trace formatter is to interpret the data that has been collected. This is described in “Syntax for Stanzas in the trace Format File” on page 15-98.



Trace Formatting

The **trcrpt** command provides a general purpose report facility. The report facility provides little data reduction, but converts the raw binary event stream to a readable ASCII listing of the event stream. Data can be visually extracted by a reader, or tools can be developed to further reduce the data.

For an event to be traced, you must write an *event hook* (sometimes called a *trace hook*) into the code that you want to trace. Tracing can be done on either the system channel (channel 0) or on a generic channel (channels 1–7). All preshipped trace points are output to the system channel.

Usually, when you want to show interaction with other system routines, use the system channel. The generic channels are provided so that you can control how much data is written to the trace log. Only your data is written to one of the generic channels.

For more information on trace hooks, see “Macros for Recording trace Events” on page 15-96.

Using the trace Facility

The following sections describe the use of the **trace** facility.

Configuring and Starting trace Data Collection

The **trace** command configures the trace facility and starts data collection. The syntax of this command is:

trace [**–a** | **–f** | **–l**] [**–c**] [**–d**] [**–h**] [**–j** *Event* [*,Event*]] [**–k** *Event* [*,Event*]] [**–m** *Message*] [**–n**] [**–o** *OutName*] [**–o–**] [**–s**] [**–L** *Size*] [**–T** *Size*] [–**1234567**]

The various options of the **trace** command are:

–f or –l Control the capture of trace data in system memory. If you specify neither the **–f** nor **–l** option, the trace facility creates two buffer areas in system memory to capture the trace data. These buffers are alternately written to the log file (or standard output if specified) as they become full. The **–f** or **–l** flag provides you with the ability to prevent data from being written to the file during data collection. The options are to collect data only until the memory buffer becomes full (**–f** for first), or to use the memory buffer as a circular buffer that captures only the last set of events that occurred before **trace** was terminated (**–l**). The **–f** and **–l** options are mutually exclusive.

With either the **–f** or **–l** option, data is not transferred from the memory collection buffers to file until **trace** is terminated.

–a Run the **trace** collection asynchronously (as a background task), returning a normal command line prompt. Without this option, the **trace** command runs in a subcommand mode (similar to the **crash** command) and returns a > prompt. You can issue subcommands and regular shell commands from the **trace** subcommand mode by preceding the shell commands with an ! (exclamation point).

–c Saves the previous trace log file adding **.old** to its name. Generates an error if a previous trace log file does not exists. When using the **–o** *Name* flag, the user-defined trace log file is renamed.

–d Delay data collection. The trace facility is only configured. Data collection is delayed until one of the collection trigger events occurs. Various methods for triggering data collection on and off are provided. These include the following:

- **trace** subcommands
- **trace** commands
- **ioctls** to /**dev**/**systrctl**

–j *Event* **or –k** *Event*
Specifies a set of events to include (**–j**) or exclude (**–k**) from the collection process. The *Event* list items can be separated by commas, or enclosed in double quotation marks and separated by commas or blanks.

–s Terminate **trace** data collection if the **trace** log file reaches its maximum specified size. The default without this option is to wrap and overwrite the data in the log file on a FIFO basis.

–h Do not write a **date**/**sysname**/**message** header to the **trace** log file.

–m *Message* Specify a text string (message) to be included in the **trace** log header record. The message is included in reports generated by the **trcrpt** command.

–n Adds some information to the trace log header: lock information, hardware information, and, for each loader entry, the symbol name, address, and type.

–o *Outfile* Specify a file to use as the log file. If you do not use the **–o** option, the default log file is /**usr**/**adm**/**ras**/**trcfile**. To direct the trace data to standard output, code the **–o** option as **–o –**. (When **–o–** is specified the **–c** flag is ignored.) Use this technique only to pipe the data stream to another process since the trace data contains raw binary events that are not displayable.

–1234567 Duplicate the **trace** design for multiple channels. Channel 0 is the default channel and is always used for recording system events. The other channels are generic channels, and their use is not predefined. There are various uses of generic channels in the system. The generic channels are also available to user applications. Each created channel is a separate events data stream. Events recorded to channel 0 are mixed with the predefined system events data stream. The other channels have no predefined use and are assigned generically.

A program can request that a generic channel be opened by using the **trcstart** subroutine. A channel number is returned, similar to the way a file descriptor is returned when it opens a file. The program can record events to this channel and, thus, have a private data stream. The **trace** command allows a generic channel to be specifically configured by defining the channel number with this option. However, this is not generally the way a

generic channel is started. It is more likely to be started from a program using the **trcstart** subroutine, which uses the returned channel ID to record events.

–T *Size* and **–L** *Size*
Specify the size of the collection memory buffers and the maximum size of the log file in bytes. The trace facility pins the data collection buffers, making this amount of memory unavailable to the rest of the system. It is important to be aware of this, because it means that the trace facility can impact performance in a memory constrained environment. If the application being monitored is not memory constrained, or if the percentage of memory consumed by the trace routine is small compared to what is available in the system, the impact of **trace** "stolen" memory should be small.

If you do not specify a value, trace uses a default size. The trace facility pins a little more than the specified buffer size. This additional memory is required for the trace facility itself. Trace pins a little more than the amount specified for first buffer mode (**–f** option). Trace pins a little more than twice the amount specified for trace configured in alternate buffer or last (circular) buffer mode.

You can also start **trace** from a the command line or with a **trcstart** subroutine call. The **trcstart** subroutine is in the **librts.a** library. The syntax of the **trcstart** subroutine is:

```
int trcstart(char *args)
```

where *args* is simply the options list desired that you would enter using the trace command if starting a system trace (channel 0). If starting a generic trace, include a **–g** option in the *args* string. On successful completion, **trcstart** returns the channel ID. For generic tracing this channel ID can be used to record to the private generic channel.

For an example of the **trcstart** routine, see the sample code on page 15-92.

When compiling a program using this subroutine, you must request the link to the **librts.a** library. Use **–l rts** as a compile option.

Controlling trace

Once **trace** is configured by the **trace** command or the **trcstart** subroutine, controls to **trace** trigger the collection of data on, trigger the collection of data off, and stop the trace facility (stop deconfigures **trace** and unpins buffers). These basic controls exist as subcommands, commands, subroutines, and ioctl controls to the **trace** control device, /**dev**/**systrctl**. These controls are described in the following sections.

Controlling trace in Subcommand Mode

If the **trace** routine is configured without the **–a** option, it runs in subcommand mode. Instead of the normal shell prompt, -> is the prompt. In this mode the following subcommands are recognized:

trcon Triggers collection of **trace** data on.

trcoff Triggers collection of **trace** data off.

q or quit Stops collection of **trace** data (like **trcoff**) and terminates **trace** (deconfigures).

!command Runs the specified shell command.

The following is an example of a trace session in which the trace subcommands are used. First, the system trace points have been displayed. Second, a trace on the system calls have been selected. Of course, you can trace on more than one trace point. Be aware that trace takes a lot of data. Only the first few lines are shown in the following example:

```
# trcrpt -j |pg
004     TRACEID IS ZERO
100     FLIH
200     RESUME
102     SLIH
103     RETURN FROM SLIH
101     SYSTEM CALL
104     RETURN FROM SYSTEM CALL
106     DISPATCH
10C     DISPATCH IDLE PROCESS
11F     SET ON READY QUEUE
134     EXEC SYSTEM CALL
139     FORK SYSTEM CALL
107     FILENAME TO VNODE (lookuppn)
15B     OPEN SYSTEM CALL
130     CREAT SYSTEM CALL
19C     WRITE SYSTEM CALL
163     READ SYSTEM CALL
10A     KERN_PFS
10B     LVM BUF STRUCT FLOW
116     XMALLOC size,align,heap
117     XMFREE address,heap
118     FORKCOPY
11E     ISSIG
169     SBREAK SYSTEM CALL


# trace -d -j 101 -m "system calls trace example"
-> trcon
-> !cp /tmp/xbugs .
-> trcoff
-> quit
# trcrpt -O "exec=on,pid=on" > cp.trace
# pg cp.trace
pr  3 11:02:02 1991
System: AIX smiller Node: 3
Machine: 000247903100
Internet Address: 00000000 0.0.0.0
system calls trace example
trace -d -j 101 -m -m system calls trace example

ID  PROCESS NAME PID  I ELAPSED_SEC  DELTA_MSEC APPL SYSCALL

001 trace       13939   0.000000000 0.000000     TRACE ON chan 0
101 trace       13939   0.000251392 0.251392     kwritev
101 trace       13939   0.000940800 0.689408     sigprocmask
101 trace       13939   0.001061888 0.121088     kreadv
101 trace       13939   0.001501952 0.440064     kreadv
101 trace       13939   0.001919488 0.417536     kioctl
101 trace       13939   0.002395648 0.476160     kreadv
101 trace       13939   0.002705664 0.310016     kioctl
```

Controlling the trace Facility by Commands

If you configure the **trace** routine to run asynchronously (the **–a** option), you can control the trace facility with the following commands:

trcon Triggers collection of trace data on.

trcoff Triggers collection of trace data off.

trcstop Stops collection of trace data (like **trcoff**) and terminates the **trace** routine.

Controlling the trace Facility by Subroutines

The controls for the **trace** routine are available as subroutines from the **librts.a** library. The subroutines return zero on successful completion. The subroutines are:

trcon Triggers collection of **trace** data on.

trcoff Triggers collection of **trace** data off.

trcstop Stops collection of **trace** data (like **trcoff**) and terminates the **trace** routine.

Controlling the trace Facility with ioctls Calls

The subroutines for controlling **trace** open the trace control device (/**dev**/**systrctl**), issue the appropriate **ioctl** command, close the control device and return. To control tracing around sections of code, it can be more efficient for a program to issue the **ioctl** controls directly. This avoids the unnecessary, repetitive opening and closing of the trace control device, at the expense of exposing some of the implementation details of **trace** control. To use the **ioctl** call in a program, include **sys**/**trcctl.h** to define the **ioctl** commands. The syntax of the **ioctl** is as follows:

```
ioctl (fd, CMD, Channel)
```

where:

fd File descriptor returned from opening /**dev**/**systrctl**

CMD TRCON, TRCOFF, or TRCSTOP

Channel Trace channel (0 for system trace).

The following code sample shows how to start a **trace** from a program and only trace around a specified section of code:

```
#include <sys/trcctl.h>
extern int trcstart(char *arg);
char *ctl_dev =”/dev/systrctl”;
int ctl_fd
main()
{
        printf(”configuring trace collection \n”);
        if (trcstart(”-ad”)){
                perror(”trcstart”);
                exit(1);
        }
        if((ctl_fd =open (ctl_dev))<0){
                perror(”open ctl_dev”);
                exit(1);
        }
        printf(”turning trace collection on \n”);
        if(ioctl(ctl_fd,TRCON,0)){
                perror(”TRCON”);
                exit(1);
        }
        /* code between here and trcoff ioctl will be traced */
        printf(”turning trace off\n”);
        if (ioctl(ctl_fd,TRCOFF,0)){
                perror(”TRCOFF”);
                exit(1);
        }
        exit(0);
}
```

Producing a trace Report

A trace report facility formats and displays the collected event stream in readable form. This report facility displays text and data for each event according to rules provided in the trace format file. The default trace format file is /**etc**/**trcfmt** and contains a stanza for each event ID. The stanza for the event provides the report facility with formatting rules for that event. This technique allows you to add your own events to programs and insert corresponding event stanzas in the format file to have their new events formatted.

This report facility does not attempt to extract summary statistics (such as CPU utilization and disk utilization) from the event stream. This can be done in several other ways. To create simple summaries, consider using **awk** scripts to process the output obtained from the **trcrpt** command.

The trcrpt Command

The syntax of the **trcrpt** command is as follows:

trcrpt [**–c**] [**–d** *List*] [**–e** *Date*] [**–h**] [**–j**] [**–k** *List*] [**–n** *Name*] [**–o** *File*] [**–p** *List*] [**–q**] [**–r**] [**–s** *Date*] [**–t** *File*] [**–v**] [**–O** *Options*] [**–T** *List*] [*LogFile*]

Normally the **trcrpt** output goes to standard output. However, it is generally more useful to redirect the report output to a file. The options are:

–c Causes the **trcrpt** command to check the syntax of the trace format file. The trace format file checked is either the default (/**etc**/**trcfmt**) or the file specified by the **–t** flag with this command. You can check the syntax of the new or modified format files with this option before attempting to use them.

–d *List* Allows you to specify a list of events to be included in the **trcrpt** output. This is useful for eliminating information that is superfluous to a given analysis and making the volume of data in the report more manageable. You may have commonly used event profiles, which are lists of events that are useful for a certain type of analysis.

–e *Date* Ends the report time with entries on, or before the specified date. The *Date* parameter has the form *mmddhhmmssyy* (month, day, hour, minute, second, and year). Date and time are recorded in the trace data only when trace data collection is started and stopped. If you stop and restart trace data collection multiple times during a trace session, date and time are recorded each time you start or stop a trace data collection. Use this flag in combination with the **–s** flag to limit the trace data to data collected during a certain time interval.

–h Omit the column headings of the report.

–j Causes the **trcrpt** command to produce a list of all the defined events from the specified trace format file. This option is useful in creating an initial file that you can edit to use as an include or exclude list for the **trcrpt** or **trace** command.

–k *List* Similar to the **–d** flag, but allows you to specify a list of events to exclude from the **trcrpt** output.

–n *Name* Specifies the kernel name list file to be used by **trcrpt** to convert kernel addresses to routine names. If not specified, the report facility uses the symbol table in /**unix**. A kernel name list file that matches the system the data was collected on is necessary to produce an accurate trace report. You can create such a file for a given level of system with the **trcnm** command:

```
trcnm /unix > Name
```

–o *File* Writes the report to a file instead of to standard output.

–p *List* Limits the **trcrpt** output to events that occurred during the running of specific processes. List the processes by process name or process ID.

–q Suppresses detailed output of syntax error messages. This is not an option you typically use.

–r Produces a raw binary format of the trace data. Each event is output as a record in the order of occurrence. This is not necessarily the order in which the events are in the trace log file since the logfile can wrap. If you use this option, direct the output to a file (or process), since the binary form of the data is not displayable.

–t *File* Allows you to specify a trace format file other than the default (/**etc**/**trcfmt**).

–T *List* Limits the report to the kernel thread IDs specified by the *List* parameter. The list items are kernel thread IDs separated by commas. Starting the list with a kernel thread ID limits the report to all kernel thread IDs *in* the list. Starting the list with a **!** (exclamation point) followed by a kernel thread ID limits the report to all kernel thread IDs *not in* the list.

–O *options* Allows you to specify formatting options to the **trcrpt** command in a comma separated list. Do not put spaces after the commas. These options take the form of option=selection. If you do not specify a selection, the command uses the default selection. The possible options are discussed in the following sections. Each option is introduced by showing its default selection.

2line=off This option lets the user specify whether the lines in the event report are split and displayed across two lines. This is useful when more columns of information have been requested than can be displayed on the width of the output device.

cpuid=off Lets you specify whether to display a column that contains the physical processor number.

endtime=nnn.nnnnnnnnn
The **starttime** and **endtime** option permit you to specify an elapsed time interval in which the **trcrpt** produces output. The elapsed time interval is specified in seconds with nanosecond resolution.

exec=off Lets you specify whether a column showing the path name of the current process is displayed. This is useful in showing what process (by name) was active at the time of the event. You typically want to specify this option. We recommend that you specify **exec=on** and **pid=on**.

ids=on Lets you specify whether to display a column that contains the event IDs. If the selection is on, a three-digit hex ID is shown for each event. The alternative is off.

pagesize=0 Lets you specify how often the column headings is reprinted. The default selection of 0 displays the column headings initially only. A selection of 10 displays the column heading every 10 lines.

pid=off Lets you specify whether a column showing the process ID of the current process is displayed. It is useful to have the process ID displayed to distinguish between several processes with the same executable name. We recommend that you specify **exec=on** and **pid=on**.

starttime=nnn.nnnnnnnnn
The **starttime** and **endtime** option permit you to specify an elapsed time interval in which the **trcrpt** command produces output. The elapsed time interval is specified in seconds with nanosecond resolution.

svc=off Lets you specify whether the report should contain a column that indicates the active system call for those events that occur while a system call is active.

tid=off Lets you specify whether a column showing the thread ID of the current thread is displayed. It is useful to have the thread ID displayed to distinguish between several threads within the same process. Alternatively, you can specify **tid=on**.

timestamp=0
The report can contain two time columns. One column is elapsed time since the **trace** command was initiated. The other possible time column is the delta time between adjacent events. The option controls if and how these times are displayed. The selections are:

- **0** Provides both an elapsed time from the start of **trace** and a delta time between events. The elapsed time is shown in seconds and the delta time is shown in milliseconds. Both fields show resolution to a nanosecond. This is the default value.
- **1** Provides only an elapsed time column displayed as seconds with resolution shown to microseconds.
- **2** Provides both an elapsed time and a delta time column. The elapsed time is shown in seconds with nanosecond resolution, and delta time is shown in microseconds with microsecond resolution.
- **3** Omits all time stamps from the report.

logfile The **logfile** is the name of the file that contains the event data to be processed by the **trcrpt** command. The default is the **/usr/adm/ras/trcfile** file.

Defining trace Events

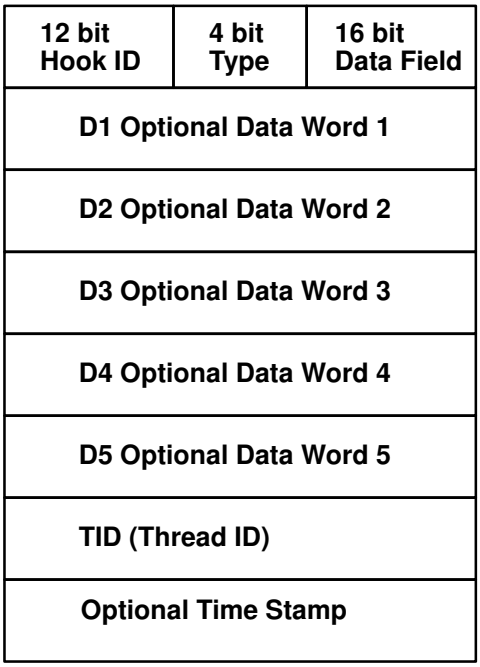
The operating system is shipped with predefined trace hooks (events). You need only activate **trace** to capture the flow of events from the operating system. You may want to define trace events in your code during development for tuning purposes. This provides insight into how the program is interacting with the system. The following sections provide the information that is required to do this.

Possible Forms of a trace Event Record

A trace event can take several forms. An event consists of a

- Hookword
- Data words (optional)
- A TID, or thread identifier
- Timestamp (optional)

The following Format of a Trace Event Record figure illustrates a trace event. A four-bit type is defined for each form the event record can take. The type field is imposed by the recording routine so that the report facility can always skip from event to event when processing the data, even if the formatting rules in the trace format file are incorrect or missing for that event.



Format of a Trace Event Record

An event record should be as short as possible. Many system events use only the hookword and timestamp. There is another event type you should seldom use because it is less efficient. It is a long format that allows you to record a variable length of data. In this long form, the 16-bit data field of the hookword is converted to a *length* field that describes the length of the event record.

Macros for Recording trace Events

There is a macro to record each possible type of event record. The macros are defined in the **sys/trcmacros.h** header file. The event IDs are defined in the **sys/trchkid.h** header file. Include these two header files in any program that is recording **trace** events. The macros to record system (channel 0) events with a time stamp are:

- **TRCHKL0T** (hw)
- **TRCHKL1T** (hw,D1)
- **TRCHKL2T** (hw,D1,D2)
- **TRCHKL3T** (hw,D1,D2,D3)
- **TRCHKL4T** (hw,D1,D2,D3)
- **TRCHKL5T** (hw,D1,D2,D3,D4,D5)

Similarly, to record non-time stamped system events (channel 0), use the following macros:

- **TRCHKL0** (hw)
- **TRCHKL1** (hw,D1)
- **TRCHKL2** (hw,D1,D2)
- **TRCHKL3** (hw,D1,D2,D3)
- **TRCHKL4** (hw,D1,D2,D3,D4)
- **TRCHKL5** (hw,D1,D2,D3,D4,D5)

There are only two macros to record events to one of the generic channels (channels 1–7). These are:

- **TRCGEN** (ch,hw,d1,len,buf)
- **TRCGENT** (ch,hw,d1,len,buf)

These macros record a hookword (hw), a data word (d1) and a length of data (len) specified in bytes from the user's data segment at the location specified (buf) to the event stream specified by the channel (ch).

Use of Event IDs (hookids)

Event IDs are 12 bits (or 3-digit hexadecimal), for a possibility of 4096 IDs. Event IDs that are permanently left in and shipped with code need to be permanently assigned. Permanently assigned event IDs are defined in the **sys/trchkid.h** header file.

To allow you to define events in your environments or during development, a range of event IDs exist for temporary use. The range of event IDs for temporary use is hex 010 through hex 0FF. No permanent (shipped) events are assigned in this range. You can freely use this range of IDs in your own environment. If you do use IDs in this range, do not let the code leave your environment.

Permanent events must have event IDs assigned by the current owner of the trace component. You should conserve event IDs because they are limited. Event IDs can be extended by the data field. The only reason to have a unique ID is that an ID is the level at which collection and report filtering is available in the trace facility. An ID can be collected or not collected by the trace collection process and reported or not reported by the trace report facility. Whole applications can be instrumented using only one event ID. The only restriction is that the granularity on choosing visibility is to choose whether events for that application are visible.

A new event can be formatted by the trace report facility (**trcrpt** command) if you create a stanza for the event in the trace format file. The trace format file is an editable ASCII file. The syntax for a format stanzas is shown in “Syntax for Stanzas in the trace Format File” on page 15-98. All permanently assigned event IDs should have an appropriate stanza in the default trace format file shipped with the base operating system.

Suggested Locations and Data for Permanent Events

The intent of permanent events is to give an adequate level of visibility to determine execution, and data flow, and have an adequate accounting for how CPU time is being consumed. During code development, it can be desirable to make very detailed use of trace for a component. For example, you can choose to trace the entry and exit of every subroutine in order to understand and tune pathlength. However, this would generally be an excessive level of instrumentation to ship for a component.

We suggest that you consult a performance analyst for decisions regarding what events and data to capture as permanent events for a new component. The following paragraphs provide some guidelines for these decisions.

Events should capture execution flow and data flow between major components or major sections of a component. For example, there are existing events that capture the interface between the virtual memory manager and the logical volume manager. If work is being queued, data that identifies the queued item (a handle) should be recorded with the event. When a queue element is being processed, the “dequeue” event should provide this identifier as data also, so that the queue element being serviced is identified.

Data or requests that are identified by different handles at different levels of the system should have events and data that allow them to be uniquely identified at any level. For example, a read request to the physical file system is identified by a file descriptor and a current offset in the file. To a virtual memory manager, the same request is identified by a segment ID and a virtual page address. At the disk device driver level, this request is identified as a pointer to a structure which contains pertinent data for the request.

The file descriptor or segment information is not available at the device driver level. Events must provide the necessary data to link these identifiers so that, for example, when a disk interrupt occurs for incoming data, the identifier at that level (which can simply be the buffer address for where the data is to be copied) can be linked to the original user request for data at some offset into a file.

Events should provide visibility to major protocol events such as requests, responses, acknowledgements, errors, and retries. If a request at some level is fragmented into multiple requests, a trace event should indicate this and supply linkage data to allow the multiple requests to be tracked from that point. If multiple requests at some level are coalesced into

a single request, a trace event should also indicate this and provide appropriate data to track the new request.

Use events to give visibility to resource consumption. Whenever resources are claimed, returned, created or deleted an event should record the fact. For example, claiming or returning buffers to a buffer pool or growing or shrinking the number of buffers in the pool.

The following guidelines can help you determine where and when you should have trace hooks in your code:

- Tracing entry and exit points of every function is not necessary. Provide only key actions and data.
- Show linkage between major code blocks or processes.
- If work is queued, associate a name (handle) with it and output it as data.
- If a queue is being serviced, the trace event should indicate the unique element being serviced.
- If a work request or response is being referenced by different handles as it passes through different software components, trace the transactions so the action or receipt can be identified.
- Place trace hooks so that requests, responses, errors, and retries can be observed.
- Identify when resources are claimed, returned, created, or destroyed.

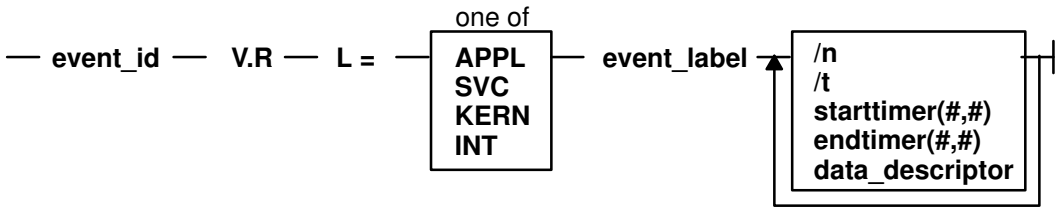
Also note that:

- A trace ID can be used for a group of events by “switching” on one of the data fields. This means that a particular data field can be used to identify from where the trace point was called. The trace format routine can be made to format the trace data for that unique trace point.
- The trace hook is the level at which a group of events can be enabled or disabled.

Syntax for Stanzas in the trace Format File

The intent of the trace format file is to provide rules for presentation and display of the expected data for each event ID. This allows new events to be formatted without changing the report facility. Rules for new events are simply added to the format file. The syntax of the rules provide flexibility in the presentation of the data.

Refer to the **/etc/tcrfmt** file to see examples of the syntax for stanzas that appear in the trace format file.



Syntax of a Stanza in the Format File

A trace format stanza can be as long as required to describe the rules for any particular event. The stanza can be continued to the next line by terminating the present line with a \ (backslash) character. The fields are:

event_id Each stanza begins with the three-digit hexadecimal event ID that the stanza describes, followed by a space.

V.R This field describes the version (V) and release (R) that the event was first assigned. Any integers work for V and R, and you may want to keep your own tracking mechanism.

L= The text description of an event can begin at various indentation levels. This improves the readability of the report output. The indentation levels correspond to the level at which the system is running. The recognized levels are:

APPL	Application level
SVC	Transitioning system call
KERN	Kernel level
INT	Interrupt

event_label The *event_label* is an ASCII text string that describes the overall use of the event ID. This is used by the **–j** option of the **trcrpt** command to provide a listing of events and their first level description. The event label also appears in the formatted output for the event unless the `event_label` field starts with an @ character.

\n The event stanza describes how to parse, label and present the data contained in an event record. You can insert a **\n** (newline) in the event stanza to continue data presentation of the data on a new line. This allows the presentation of the data for an event to be several lines long.

\t The **\t** (tab) function inserts a tab at the point it is encountered in parsing the description. This is similar to the way the **\n** function inserts new lines. Spacing can also be inserted by spaces in the `data_label` or `match_label` fields.

starttimer(#,#)

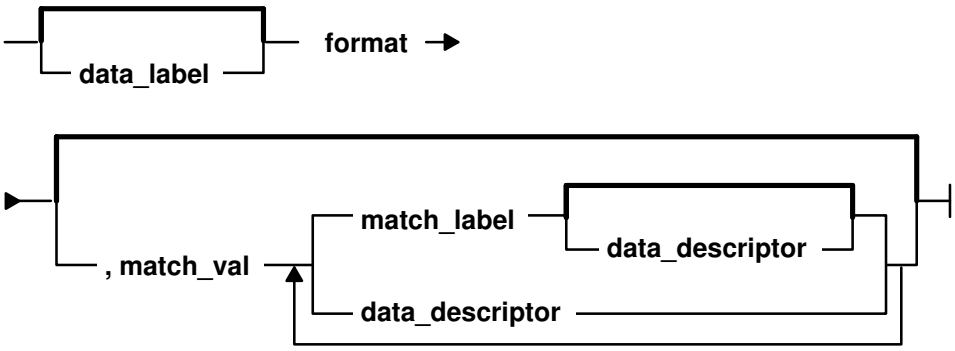
The `starttimer` and `endtimer` fields work together. The (#,#) field is a unique identifier that associates a particular `starttimer` value with an `endtimer` that has the same identifier. By convention, if possible, the identifiers should be the ID of starting event and the ID of the ending event.

When the report facility encounters a start timer directive while parsing an event, it associates the starting events time with the unique identifier. When an end timer with the same identifier is encountered, the report facility outputs the delta time (this appears in brackets) that elapsed between the starting event and ending event. The begin and end system call events make use of this capability. On the return from system call event, a delta time is shown that indicates how long the system call took.

endtimer(#,#) See the `starttimmer` field in the preceding paragraph.

data_descriptor

The `data_descriptor` field is the fundamental field that describes how the report facility consumes, labels, and presents the data. The following Syntax of the data_descriptor Field figure illustrates this field's syntax.



Syntax of the data_descriptor Field

The various subfields of the `data_descriptor` field are:

data_label The data label is an ASCII string that can optionally precede the output of data consumed by the following `format` field.

format Review the format of an event record depicted in the figure Format of a trace Event Record. You can think of the report facility as having a pointer into the data portion of an event. This data pointer is initialized to point to the beginning of the event data (the 16-bit data field in the hookword). The `format` field describes how much data the report facility consumes from this point and how the data is considered. For example, a value of **Bm.n** tells the report facility to consume m bytes and n bits of data and to consider it as binary data.

The possible `format` fields are described in the following section. If this field is not followed by a comma, the report facility outputs the consumed data in the format specified. If this field is followed by a comma, it signifies that the data is not to be displayed but instead compared against the following `match_vals` field. The data descriptor associated with the matching `match_val` field is then applied to the remainder of the data.

match_val The match value is data of the same format described by the preceding format fields. Several match values typically follow a format field that is being matched. The successive match fields are separated by commas. The last match value is not followed by a comma. Use the character string * as a pattern-matching character to match anything. A pattern-matching character is frequently used as the last element of the `match_val` field to specify default rules if the preceding `match_val` field did not occur.

match_label The match label is an ASCII string that is output for the corresponding match.

Each of the possible `format` fields is described in the comments of the **/etc/trcfmt** file. The following shows several possibilities:

Format field	**descriptions**
Am.n	This value specifies that m bytes of data are consumed as ASCII text, and that it is displayed in an output field that is n characters wide. The data pointer is moved m bytes.
S1, S2, S4	Left justified string. The length of the field is defined as 1 byte (S1), 2 bytes (S2), or 4 bytes (S4). The data pointer is moved accordingly.
Bm.n	Binary data of m bytes and n bits. The data pointer is moved accordingly.
Xm	Hexadecimal data of m bytes. The data pointer is moved accordingly.
D2, D4	Signed decimal format. Data length of 2 (D2) bytes or 4 (D4) bytes is consumed.
U2, U4	Unsigned decimal format. Data length of 2 or 4 bytes is consumed.
F4, F8	Floating point of 4 or 8 bytes.
Gm.n	Positions the data pointer. It specifies that the data pointer is positioned m bytes and n bits into the data.
Om.n	Skip or omit data. It omits m bytes and n bits.
Rm	Reverse the data pointer m bytes.

Some macros are provided that can be used as format fields to quickly access data. For example:

$D1, $D2, $D3, $D4, $D5
: These macros access data words 1 through 5 of the event record without moving the data pointer. The data accessed by a macro is hexadecimal by default. A macro can be cast to a different data type (X, D, U, B) by using a % character followed by the new format code. For example, the following macro causes data word one to be accessed, but to be considered as 2 bytes and 3 bits of binary data:

```
$D1%B2.3
```

$HD
: This macro accesses the first 16 bits of data contained in the hookword, in a similar manner as the $D1 through $D5 macros access the various data words. It is also considered as hexadecimal data, and also can be cast.

You can define other macros and use other formatting techniques in the trace format file. This is shown in the following trace format file example.

Example Trace Format File

```
# .
# .
# .
#
# I. General Information
#
# A. Binary format for the tracehook calls. (1 column = 4 bits)
#    trchk    MMmTDDDD
#    trchkt   MMmTDDDDtttttttt
#    trchkl   MMmTDDDD11111111
#    trchklt  MMmTDDDD11111111tttttttt
#    trchkg   MMmTDDDD11111111222222223333333344444444555555555
#    trchkg   MMmTDDDD11111111222222223333333344444444555555555tttttttt
#    trcgen   MMmTLLLL11111111vvvvvvvvvvvvvvvvvvvvvvvvvvvvxxxxxx
#    trcgent  MMmTLLLL11111111vvvvvvvvvvvvvvvvvvvvvvvvvvvvxxxxxxtttttttt
#
#         legend:
#    MM = major id
#    m  = minor id
#    T  = hooktype
#    D  = hookdata
#    t  = nanosecond timestamp
#    1  = d1 (see trchkid.h for calling syntax for the tracehook routines)
#    2  = d2, etc.
#    v  = trcgen variable length buffer
#    L  = length of variable length data in bytes.
#
#  The DATA_POINTER starts at the third byte in the event, ie.,
#    at the 16 bit hookdata DDDD.
#  The trcgen() type (6,7) is an exception.The DATA_POINTER starts at
#    the fifth byte, ie., at the 'd1' parameter 11111111.
#
# B. Indentation levels
#    the left margin is set per template using the 'L=XXXX' command.
#    The default is L=KERN, the second column.
#    L=APPL moves the left margin to the first  column.
#    L=SVC  moves the left margin to the second column.
#    L=KERN moves the left margin to the third  column.
#    L=INT  moves the left margin to the fourth column.
#    The command if used must go just after the version code.
#
#    Example usage:
#113 1.7 L=INT "stray interrupt" ... \
#
# C. Continuation code and delimiters.
#    A '\' at the end of the line must be used to continue the template
#     on the next line.
#    Individual strings (labels) can be separated by one or more blanks
#     or tabs. However, all whitespace is squeezed down to 1 blank on
#     the report. Use '\t' for skipping to the next tabstop, or use
#     A0.X format (see below) for variable space.
#
#
# II. FORMAT codes
#
# A. Codes that manipulate the DATA_PONTER
# Gm.n
#     "Goto"    Set DATA_POINTER to byte.bit location m.n
#
# Om.n
#     "Omit"    Advance DATA_POINTER by m.n byte.bits
#
# Rm
```

```
#     ”Reverse” Decrement DATA_POINTER by m bytes. R0 byte aligns.
#
# B. Codes that cause data to be output.
# Am.n
#     Left justified ascii.
#     m=length in bytes of the binary data.
#     n=width of the displayed field.
#     The data pointer is rounded up to the next byte boundary.
#     Example
#      DATA_POINTER|
#                  V
#             xxxxxhello world\0xxxxxx
#
#  i.   A8.16 results in:                        |hello wo        |
#       DATA_POINTER--------|
#                           V
#             xxxxxhello world\0xxxxxx
#
#  ii.  A16.16 results in:                       |hello world     |
#       DATA_POINTER----------------|
#                                   V
#             xxxxxhello world\0xxxxxx
#
#  iii. A16 results in:                          |hello world|
#       DATA_POINTER----------------|
#                                   V
#             xxxxxhello world\0xxxxxx
#
#  iv.  A0.16 results in:                        |               |
#       DATA_POINTER|
#                    V
#             xxxxxhello world\0xxxxxx
#
# S1, S2, S4
# Left justified ascii string.
# The length of the string is in the first byte(half-word, word)
# of the data. This length of the string does not include this byte.
#  The data pointer is advanced by the length value.
#     Example
#      DATA_POINTER|
#                  V
#             xxxxxBhello worldxxxxxx     (B = hex 0x0b)
#
#  i.   S1 results in:                           |hello world|
#       DATA_POINTER-----------|
#                              V
#             xxxxBhello worldxxxxxx
#
# $reg%S1
#     A register with the format code of ’Sx’ works ”backwards”
#     from a register with a different type. The format is Sx,
#     but the length of the string comes from $reg instead of the
#     next n bytes.
#
# Bm.n
#     Binary format.
#     m = length in bytes
#     n = length in bits
#     The length in bits of the data is m * 8 + n. B2.3 and B0.19
#     are the same. Unlike the other printing FORMAT codes, the
#     DATA_POINTER can be bit aligned and is not rounded up to
#     the next byte boundary.
#
# Xm
```

```
#     Hex format.
#     m = length in bytes. m=0 thru 16
#     The DATA_POINTER is advanced by m.
#
# D2, D4
#     Signed decimal format.
#     The length of the data is 2 (4) bytes.
#     The DATA_POINTER is advanced by 2 (4).
#
# U2, U4
#     Unsigned decimal format.
#     The length of the data is 2 (4) bytes.
#     The DATA_POINTER is advanced by 2 (4).
#
# F4
#     Floating point format. (like %0.4E)
#     The length of the data is 4 bytes.
#     The format of the data is that of C type 'float'.
#     The DATA_POINTER is advanced by 4.
#
# F8
#     Floating point format. (like %0.4E)
#     The length of the data is 8 bytes.
#     The format of the data is that of C type 'double'.
#     The DATA_POINTER is advanced by 8.
#
# HB
#     Number of bytes in trcgen() variable length buffer.
#     This is also equal to the 16 bit hookdata.
#     The DATA_POINTER is not changed.
#
# HT
#  The hooktype. (0 - E)
#  trcgen  = 0, trchk  = 1, trchl  = 2, trchkg  = 6
#  trcgent = 8, trchkt = 9, trchlt = A, trchkgt = E
#  HT & 0x07 masks off the timestamp bit
#  This is used for allowing multiple, different trchkx() calls with
#       the same template.
#     The DATA_POINTER is not changed.
#
# C. Codes that interpret the data in some way before output.
# T4
#     Output the next 4 bytes as a data and time string,
#     in GMT timezone format.  (as in ctime(&seconds))
#     The DATA_POINTER is advanced by 4.
#
# E1,E2,E4
#     Output the next byte (half_word, word) as an 'errno' value,
#     replacing the numeric code with the corresponding #define name in
#     /usr/include/sys/errno.h
#     The DATA_POINTER is advanced by 1, 2, or 4.
#
# P4
#  Use the next word as a process id (pid), and output the
#  pathname of the executable with that process id.Process
#  ids and their pathnames are acquired by the trace command at
#  the start of a trace and by trcrpt via a special EXEC tracehook.
#  The DATA_POINTER is advanced by 4.
#
# \t
#  Output a tab. \t\t\t outputs 3 tabs. Tabs are expanded to spaces,
#  using a fixed tabstop separation of 8.If L=0 indentation is used,
#     the first tabstop is at 3.
#     The DATA_POINTER advances over the \t.
```

```
#
# \n
#  Output a newline. \n\n\n outputs 3 newlines.
#  The newline is left-justified according to the INDENTATION LEVEL.
#     The DATA_POINTER advances over the \n.
#
# $macro
#     The value of 'macro' is output as a %04X value. Undefined
#     macros have the value of 0000.
#     The DATA_POINTER is not changed.
#     An optional format can be used with macros:
#        $v1%X4    will output the value $v1 in X4 format.
#        $zz%B0.8  will output the value $v1 in 8 bits of binary.
#     Understood formats are: X, D, U, B. Others default to X2.
#
# "string"     'string' data type
#  Output the characters inside the double quotes exactly. A string
#     is treated as a descriptor. Use "" as a NULL string.
#
# `string format $macro` If a string is backquoted, it is expanded
#   as a quoted string, except that FORMAT codes and $registers are
#     expanded as registers.
#
# III. SWITCH statement
#     A format code followed by a comma is a SWITCH statement.
#     Each CASE entry of the SWITCH statement consists of
#       1. a 'matchvalue' with a type (usually numeric) corresponding
#          to the format code.
#       2. a simple 'string' or a new 'descriptor' bounded by braces.
#          A descriptor is a sequence of format codes, strings,
#          switches and loops.
#       3. and a comma delimiter.
#  The switch is terminated by a CASE entry without a comma
#  delimiter.   The CASE entry is selected to as the first
#  entry whose matchvalue   is equal to the expansion of the format
#  code. The special matchvalue '\*' is a wildcard and matches
#  anything.
#  The DATA_POINTER is advanced by the format code.
#
#
# IV. LOOP statement
#     The syntax of a 'loop' is
#     LOOP format_code { descriptor }
#     The descriptor is executed N times, where N is the numeric value
#     of the format code.The DATA_POINTER is advanced by the
#     format code plus whatever the descriptor does. Loops are used to
#     output binary buffers of data, so descriptor is
#     usually simply X1 or X0. Note that X0 is like X1 but does not
#     supply a space separator ' ' between each byte.
#
# V. macro assignment and expressions
#  'macros' are temporary (for the duration of that event) variables
#   that work like shell variables.
#   They are assigned a value with the syntax:
#   {{ $xxx = EXPR }}
#   where EXPR is a combination of format codes, macros, and constants.
#   Allowed operators are + - / *
#   For example:
#{{ $dog = 7 + 6 }} {{ $cat = $dog * 2 }} $dog $cat
#
#   will output:
#000D 001A
#
#   Macros are useful in loops where the loop count is not always
```

```
#   just before the data:
#G1.5 {{ $count = B0.5 }} G11 LOOP $count {X0}
#
#   Up to 25 macros can be defined per template.
#
#
# VI. Special macros:
# $RELLINENO  line number for this event. The first line starts at 1.
# $D1 - $D5   dataword 1 through dataword 5. No change to datapointer.
# $HD         hookdata (lower 16 bits)
# $SVC        Output the name of the current SVC
# $EXECPATH   Output the pathname of the executable for current process.
# $PID        Output the current process id.
# $ERROR      Output an error message to the report and exit from the
             template after the current descriptor is processed.
#            The error message supplies the logfile, logfile offset of
#            the start of that event, and the traceid.
# $LOGIDX     Current logfile offset into this event.
# $LOGIDX0    Like $LOGIDX, but is the start of the event.
# $LOGFILE    Name of the logfile being processed.
# $TRACEID    Traceid of this event.
# $DEFAULT    Use the DEFAULT template 008
# $STOP       End the trace report right away
# $BREAK      End the current trace event
# $SKIP       Like break, but don't print anything out.
# $DATAPOINTER  The DATA_POINTER. It can be set and manipulated
#             like other user-macros.
#             {{ $DATAPOINTER = 5 }} is equivalent to G5
# $BASEPOINTER Usually 0. It is the starting offset into an event.The
#             actual offset is the DATA_POINTER + BASE_POINTER. It is used
#             with template subroutines, where the parts on an event have
#             the same structure, and can be printed by the same template,
#             but may have different starting points into an event.
#
# VII. Template subroutines
#    If a macro name consists of 3 hex digits, it is a "template
#    subroutine". The template whose traceid equals the macro name
#     is inserted in place of the macro.
#
#    The data pointer is where is was when the template
#    substitution was encountered.Any change made to the data pointer
#    by the template subroutine remains in affect when the template
#    ends.
#
#    Macros used within the template subroutine correspond to those
#    in the calling template. The first definition of a macro in the
#    called template is the same variable as the first in the called.
#    The names are not related.
#
#    Example:
#    Output the trace label ESDI STRATEGY.
#    The macro '$stat' is set to bytes 2 and 3 of the trace event.
#    Then call template 90F to interpret a buf header. The macro
#    '$return' corresponds to the macro '$rv', since they were
#    declared in the same order. A macro definition with
#     no '=' assignment just declares the name
#    like a place holder. When the template returns,the saved special
#    status word is output and the returned minor device number.
#
#900 1.0 "ESDI STRATEGY" {{ $rv = 0 }} {{ $stat = X2 }} \
#       $90F \n\
#special_esdi_status=$stat for minor device $rv
#
#90F 1.0 "" G4 {{ $return }} \
```

```
#       block number X4 \n\
#       byte count   X4 \n\
#       B0.1, 1 B_FLAG0 \
#       B0.1, 1 B_FLAG1 \
#       B0.1, 1 B_FLAG2 \
#       G16 {{ $return = X2 }}
#
#    Note: The $DEFAULT reserved macro is the same as $008
#
# VII. BITFLAGS statement
#     The syntax of a 'bitflags' is
#     BITFLAGS [format_code|register],
#          flag_value string {optional string if false},   or
#          '&' mask field_value string,
#                   ...
#
#     This statement simplifies expanding state flags, since it look
#       a lot like a series of #defines.
#     The '&' mask is used for interpreting bit fields.
#     The mask is anded to the register and the result is compared to
#       the field_value. If a match, the string is printed.
#    The base is 16 for flag_values and masks.
#    The DATA_POINTER is advanced if a format code is used.
#    Note:the default base for BITFLAGS is 16. If the mask or field
#    value has a leading 0, the number is octal. 0x or 0X makes the
#    number hex.
# A 000 traceid will use this template
# This id is also used to define most of the template functions.
#  filemode(omode) expand omode the way  ls -l does. The
#                  call to setdelim() inhibits spaces between the chars.
#
```

Examples of Coding Events and Formatting Events

There are five basic steps involved in generating a trace from your software program.

Step 1: Enable the trace

Enable and disable the trace from your software that has the trace hooks defined. The following code shows the use of trace events to time the running of a program loop.

```
#include                <sys/trcctl.h>
#include                <sys/trcmacros.h>
#include                <sys/trchkid.h>

char            *ctl_file = ”/dev/systrctl”;
int             ctlfd;
int             i;

main()
{
        printf(”configuring trace collection \n”);
        if (trcstart(”-ad”)){
                perror(”trcstart”);
                exit(1);
        }
        if((ctlfd = open(ctl_file,0))<0){
                perror(ctl_file);
                exit(1);
        }
        printf(”turning  trace on \n”);
        if(ioctl(ctlfd,TRCON,0)){
                perror(”TRCON”);
                exit(1);
        }
        /* here is the code that is being traced */
        for(i=1;i<11;i++){
                TRCHKL1T(HKWD_USER1,i);
                /*   sleep(1) */
                /* you can uncomment sleep to make the loop
                /* take longer. If you do, you will want to
                /* filter the output or you will be */
                /* overwhelmed with 11 seconds of data */
        }
        /* stop tracing code */
        printf(”turning trace off\n”);
        if(ioctl(ctlfd,TRCSTOP,0)){
                perror(”TRCOFF”);
                exit(1);
        }
        exit(0);
}
```

Step 2: Compile your program

When you compile the sample program, you need to link to the **librts.a** library:

```
cc -o sample sample.c -l rts
```

Step 3: Run the program

Run the program. In this case, it can be done with the following command:

```
./sample
```

You must have root privilege if you use the default file to collect the trace information (**/usr/adm/ras/trcfile**).

Step 4: Add a stanza to the format file

This provides the report generator with the information to correctly format your file. The report facility does not know how to format the **HKWD_USER1** event, unless you provide rules in the trace format file.

The following is an example of a stanza for the **HKWD_USER1** event. The **HKWD_USER1** event is event ID 010 hexadecimal. You can verify this by looking at the **sys/trchkid.h** header file.

```
# User event HKWD_USER1 Formatting Rules Stanza
# An example that will format the event usage of the sample program
010 1.0 L=APPL "USER EVENT - HKWD_USER1" O2.0 \n\
               "The # of loop iterations =" U4\n\
               "The elapsed time of the last loop = "\
                endtimer(0x010,0x010) starttimer(0x010,0x010)
```

Note: When entering the example stanza, do not modify the master format file **/etc/trcfmt**. Instead, make a copy and keep it in your own directory. This allows you to always have the original trace format file available.

Step 5: Run the format/filter program

Filter the output report to get only your events. To do this, run the **trcrpt** command:

```
trcrpt -d 010 -t mytrcfmt -O exec=on -o sample.rpt
```

The formatted trace results are:

```
ID    PROC NAME  I  ELAPSED_SEC  DELTA_MSEC   APPL    SYSCALL    KERNEL  INTERRUPT

010   sample       0.000105984  0.105984  USER HOOK 1
                                           The data field for the user hook = 1
010   sample       0.000113920  0.007936  USER HOOK 1
                                           The data field for the user hook = 2 [7 usec]
010   sample       0.000119296  0.005376  USER HOOK 1
                                           The data field for the user hook = 3 [5 usec]
010   sample       0.000124672  0.005376  USER HOOK 1
                                           The data field for the user hook = 4 [5 usec]
010   sample       0.000129792  0.005120  USER HOOK 1
                                           The data field for the user hook = 5 [5 usec]
010   sample       0.000135168  0.005376  USER HOOK 1
                                           The data field for the user hook = 6 [5 usec]
010   sample       0.000140288  0.005120  USER HOOK 1
                                           The data field for the user hook = 7 [5 usec]
010   sample       0.000145408  0.005120  USER HOOK 1
                                           The data field for the user hook = 8 [5 usec]
010   sample       0.000151040  0.005632  USER HOOK 1
                                           The data field for the user hook = 9 [5 usec]
010   sample       0.000156160  0.005120  USER HOOK 1
                                           The data field for the user hook = 10 [5 usec]
```

Usage Hints

The following sections provide some examples and suggestions for use of the trace facility.

Viewing trace Data

Include several optional columns of data in the trace output. This causes the output to exceed 80 columns. It is best to view the reports on an output device that supports 132 columns.

Bracketing Data Collection

Trace data accumulates rapidly. Bracket the data collection as closely around the area of interest as possible. One technique for doing this is to issue several commands on the same command line. For example, the command:

```
trace -a; cp /etc/trcfmt /tmp/junk; trcstop
```

captures the total execution of the copy command.

Note: This example is more educational if the source file is not already cached in system memory. The **trcfmt** file can be in memory if you have been modifying it or producing trace reports. In that case, choose as the source file some other file that is 50 to 100KB and has not been touched.

Reading a trace Report

The trace facility displays system activity. It is a useful learning tool to observe how the system actually performs. The previous output is an interesting example to browse. To produce a report of the copy, use the following:

```
trcrpt -O "exec=on,pid=on" > cp.rpt
```

In the **cp.rpt** file you can see the following activities:

- The fork, exec, and page fault activities of the **cp** process.
- The opening of the /**etc**/**trcfmt** file for reading and the creation of the /**tmp**/**junk** file.
- The successive **read** and **write** subroutiness to accomplish the copy.
- The **cp** process becoming blocked while waiting for I/O completion, and the wait process being dispatched.
- How logical volume requests are translated to physical volume requests.
- The files are mapped rather than buffered in traditional kernel buffers. The read accesses cause page faults that must be resolved by the virtual memory manager.
- The virtual memory manager senses sequential access and begins to prefetch the file pages.
- The size of the prefetch becomes larger as sequential access continues.
- The writes are delayed until the file is closed (unless you captured execution of the **sync** daemon that periodically forces out modified pages).
- The disk device driver coalesces multiple file requests into one I/O request to the drive when possible.

The trace output looks a little overwhelming at first. This is a good example to use as a learning aid. If you can discern the activities described, you are well on your way to being able to use the trace facility to diagnose system performance problems.

Effective Filtering of the trace Report

The full detail of the trace data may not be required. You can choose specific events of interest to be shown. For example, it is sometimes useful to find the number of times a certain event occurred. Answer the question, “how many opens occurred in the copy example?” First, find the event ID for the **open** subroutine:

```
trcrpt -j | pg
```

You can see that event ID 15b is the open event. Now, process the data from the copy example (the data is probably still in the log file) as follows:

```
trcrpt -d 15b -O "exec=on"
```

The report is written to standard output and you can determine the number of opens that occurred. If you want to see only the opens that were performed by the cp process, run the report command again using:

```
trcrpt -d 15b -p cp -O "exec=on"
```

This command shows only the opens performed by the **cp** process.

Chapter 16. Power Management (PM) Aware Device Drivers

This chapter provides information on the modification of an existing AIX device driver for Power Management.

Power Management-Aware Device Drivers: Overview

PM-aware device drivers are responsible for handling power-management-related operations for corresponding devices.

The Power Management kernel extension (PM core) is a coordinator for PM-aware device drivers. The PM core informs the device drivers of various PM events and also requests some of the drivers to enter a low power mode if they seem to be idle. Because power management features are completely dependent on each machine, the role of the PM core is to abstract the difference between the machines. The following are PM features:

device local standby
: Moves a device to a low power mode if the device is idle. In certain cases, low power mode just turns the device off locally. For example, power to a hard disk drive can be stopped using a standard or unique command if it has not been accessed for a while.

suspend
: Powers off everything on a power-managed machine except the system memory, the memory controller, and some power management-related logic. Therefore, all the volatile data in the remaining system devices is lost. All the device drivers that directly access those devices must have an appropriate routine to retrieve the data at the start of the suspend state. In certain cases, the device drivers may need to perform some operations before the corresponding device is turned off as a result of entering suspend. Since system memory, including the private memory area of the device driver, is preserved during the suspend state, each device driver can retrieve the previous device context using the data in its own private memory area.

hibernation
: During the hibernation state, all power source, except the special logic for turning the system on again, is shut off. At the restart from hibernation, the volatile data of all the devices is lost. System main memory, however, is saved or restored by the power management kernel extension using nonvolatile storage, such as the hard disk. There is no difference between the suspend and hibernation states for device drivers whose device data was lost during those power management states.

Warning: All device drivers that are to be used on a system supporting suspend and hibernation states must be PM-aware. Since, in these states, power is removed from all physical devices, it is necessary for the corresponding device drivers to be notified of state transitions to save and restore device states as necessary. A non-PM-aware device driver could experience unpredictable behavior following a suspend or hibernation and, in some cases, could abend the system, or depending on the device design, could cause physical damage to the device.

Pseudo-device drivers, drivers that do not control physical devices, might also want to be PM-aware in order to monitor power management state transitions. For example, network pseudo-drivers would be able to gracefully disconnect a network link prior to a suspend or hibernation**.**

PM Core versus PM-aware Device Driver Operations

The following is a summary of the power management operations of PM-aware device drivers as opposed to the PM core.

PM Core Notifies all registered PM-aware device drivers of power management state transitions. The state transitions can be the result of a user action, an external event, or the expiration of a system timer.

Maintains idle counts for each device in the system, as well as for the system as a whole.

PM-aware Device Driver

Handles commands from the PM core such as **device idle**, **suspend**, **hibernate**, and **enable**. The PM-aware device driver then takes device-specific action appropriate for the command. For example, if the device driver receives a **device idle** request, the driver places the device into a low power mode or even in an **off** state.

The PM-aware device driver can reject a command from the PM core if the driver is unable to perform the request. For example, if the device has an outstanding operation that does not complete for several seconds, the PM-aware device driver can reject the request from the PM core.

If there is an outstanding command that will complete within a second or two, the PM-aware device driver waits for completion of the outstanding command and returns successful to the PM core.

The PM-aware device driver maintains an activity flag to inform the PM core whether the device driver is busy.

Power Management Kernel Services

The following sections contain information on the power management kernel services. For more detailed information on the syntax and return codes of each PM kernel service, see *AIX Technical Reference, Volume 5: Kernel and Subsystems*.

pm_register_handle

The **pm_register_handle** kernel service registers and unregisters a PM handle to the PM core.

Registration order

The order of PM-aware device driver registration corresponds to the order in which the device drivers are configured. The PM core notifies registered handlers in the reverse order of registration when starting suspend or hibernation. For example, the last registered handler is the first one called. This ordering eliminates problems with parent and child device dependencies among layered device drivers. In order for a parent driver to receive notification from the PM core earlier than its children, the PM core uses the same order of registration when resuming from suspend or hibernation.

Struct pm_handle

The **pm_handle** structure is the communications vehicle between the PM-aware device driver and the PM core. The device driver is responsible for allocating the **pm_handle** structure and ensuring that it is pinned for accesses to the structure during suspend and hibernation system state transitions.

The following is the **pm_handle** structure definition:

```
struct pm_handle {
    int                 activity;                       /* PM aware DD sets this value when accesses occur */
    int                 mode;                           /* PM aware DD needs to maintain this device mode */
    int                 device_idle_time;                       /* idle timer value during system PM enable */
    int                 device_standby_time;                    /* idle timer value during system standby */
    int                 idle_counter;                           /* idle time counter */
    int                 (*handler)(caddr_t private, int ctrl); /* PM core calls this subroutine */
    caddr_t             private;                        /* private pointer passed to the handler subroutine */
    dev_t               devno;                          /* device major number minor number */
    int                 attribute;                      /* device attributes */
    struct pm_handle    *next1;                         /* next pointer used by PM core */
    struct pm_handle    *next2;                         /* next pointer used by PM core */
    int                 device_idle_time1;                      /* idle timer value for DPMS standby mode */
    int                 device_idle_time2;                      /* idle timer value for DPMS suspend mode */
    char                *device_logical_name;                   /* device logical name */
    char                reserve[2];                             /* reserved area for future use */
    ushort              pm_version;                             /* phase 1: 0x0000, phase 2: 0x0100 */
    int                 *extension;                             /* for future expandability */
}
```

The following are fields in the **pm_handle** structure:

`pm_handle.activity`

Used by a PM-aware device driver to notify the PM core of the **busy** state of the device.

For most devices, when the driver processes a request for that device, it sets the `activity` field to 1. The PM core regularly monitors the `activity` field of each registered device for determining device idle times, as well as system idle times. The PM core resets each device's `activity` field to 0 on each periodic check.

For some devices, the device driver does not get a chance to set the `activity` field to 1 even though the device is in use. Examples include a graphics adapter driver that processes requests for the device after giving the X-server direct access to the device. These devices must set the `activity` field to –1. The PM core does not reset a device's `activity` field if it is set to –1, nor does the core increment the device's idle count.

Other devices that can set the `activity` field to –1 include asynchronous interrupt processing devices such as keyboard, mouse, and serial port devices. When they receive a **suspend** or **hibernation** transition request from the PM core, these device drivers should set the `activity` field to –1. At resume time, the PM core automatically sends device PM enable requests to all drivers whose `activity` field was –1 prior to entering the **suspend** or **hibernation** state. Setting the `activity` field to –1 also indicates that no bus master I/O activity is occurring for that device.

The asynchronous interrupting devices must also disable their device interrupts once they accept a **suspend** or **hibernation** notification to preserve the current state.

The default value for the `activity` field is 0.

`pm_handle.mode`

The `mode` field is updated by the PM-aware device driver after completing a device mode transition as instructed by the PM core. The following is a description of each of the device modes:

PM_DEVICE_FULL_ON
: The normal device operational mode. The device is fully powered without any device-level power management active. At device configuration, all devices are in the full-on mode.

PM_DEVICE_ENABLE
: In the PM-enabled state, device drivers can activate device-level power management. All devices are moved into the PM-enabled state when the PM core is configured and initialized. After the device is in suspend or hibernation state, this mode can be invoked by the PM core in response to a failed hibernation or suspend transition.

PM_DEVICE_IDLE
: In PM_DEVICE_IDLE mode, the device driver causes its device to enter a power saving mode. Any access to the device brings the device driver back to the PM_DEVICE_ENABLE mode.

PM_DEVICE_SUSPEND
: In PM_DEVICE_SUSPEND mode, the device is powered off. The device driver saves necessary device state for restoring upon resume from suspend state. During this state, the device driver does not process any newly arriving I/O requests, but instead, blocks the requesting process until after the resume.

  Some devices might have downloadable microcode that will need to be restored as part of the resume from suspend or hibernation. In these cases, the device configuration method should provide the device driver with the file system path to the microcode download file at device configuration time. Then, during the resume from suspend or hibernation, the device driver can use the **fp_open( )** and **fp_read( )** kernel services to open and read in its microcode file and then download it prior to resuming normal operation.

PM_DEVICE_HIBERNATION
: In PM_DEVICE_HIBERNATION mode, the device is powered off. To the device driver, this mode is no different from the PM_DEVICE_SUSPEND mode. The device driver saves necessary device state for restoring upon resume from hibernation state. During this state, the device driver does not process any newly arriving I/O requests, but instead, blocks the requesting process until after the resume.

PM_DEVICE_DPMS_STANDBY
: This mode is used only by graphics device drivers. In PM_DEVICE_DPMS_STANDBY mode, the display is dimmed.

PM_DEVICE_DPMS_SUSPEND
: This mode is used only by graphics device drivers. In PM_DEVICE_DPMS_SUSPEND mode, the display is suspended.

The PM core updates a device's mode field in the case of a transition from **suspend** to **hibernation**. In this case, the PM core changes the mode from PM_DEVICE_SUSPEND to PM_DEVICE_HIBERNATION, so that the device driver can accurately determine the previous mode when resuming.

The default value for the `mode` field is PM_FULL_ON.

`pm_handle.device_idle_time`
: The `device_idle_time` field indicates the period of device inactivity before the PM core requests that the device go to PM_DEVICE_IDLE mode.

The value for the `device_idle_time` field is retrieved from the ODM and never changed by the device driver.

`pm_handle.device_standby_time`
: When the system is in the PM_STANDBY state, the `device_standby_time` field indicates the period of device inactivity before the PM core requests that the device go to the PM_DEVICE_IDLE state.

The value for the `device_standby_time` field is retrieved from the ODM and never changed by the device driver.

`pm_handle.idle_counter`
: The `idle_counter` field is incremented by the PM core each time a periodic check of the device's `activity` field indicates the device is idle.

The default value for the `idle_counter` is 0. This field is never modified by the device driver.

`pm_handle.handler`
: The `handler` field is set by the device driver to point to the device driver's power management handler. This function is the entry point called by the PM core to send state transition notifications to the device driver.

The device driver takes care to ensure this function is pinned prior to processing a transition to the suspend or hibernation state. Also, the driver must not call any kernel services listed as "process environment only" during the suspend or hibernation transition. In other words, the driver must ensure that no page faults occur during the suspend or hibernation transition.

The parameters to the handler function are:

private
: The **private** parameter is the `pm_handle.private` field as used by the device driver. The driver can choose to have the `private` field point to a device-specific data structure.

ctrl
: The **ctrl** parameter specifies a PM core-initiated mode transition or notification. The notifications are one of the following:

PM_PAGE_FREEZE_NOTICE. Notifies the device driver that a subsequent suspend or hibernation state transition is imminent. The device driver uses this notification to pin necessary code and data to be used during the suspend or hibernate paths. If the device driver needs to **xmalloc** or **xmfree** anything, it must be done before returning from this notification.

PM_PAGE_UNFREEZE_NOTICE. Notifies the device driver that the suspend or hibernation transition is complete and the driver can unpin relevant code and data.

PM_RING_RESUME_ENABLE_NOTICE. Notifies the device driver when the PM_RING_RESUME_SUPPORT flags are set. In response, the driver enables its device's corresponding ring resume feature.

The `pm_handle.handler` is initialized to point to the device driver's PM handler and not changed again.

`pm_handle.private`

The `private` field is available for device driver-specific use. This field is the first parameter on calls to the registered PM handler. For example, a device driver that controls multiple physical devices can register multiple times, once for each physical device. In this case, the driver uses the `private` field to distinguish between the registrations.

If unused, the `private` field is set to NULL.

`pm_handle.devno`

The device driver sets its major or minor device number in the `devno` field. If the device does not have a major or minor number, the `devno` field is set to 0. The PM core never modifies the `devno` field.

`pm_handle.attribute`

The `attribute` field is used by specific PM-aware device drivers whose activity affects the power management modes of another device.

For example, the activity of a graphics input device affects the low power mode of a graphics output device, such as keyboard input, causing a graphics screen to unblank. Also, an audio input device can affect the low power mode of an audio output device, such as CD-ROM input, causing the audio output to get out of low power mode. The following is a list of possible attribute bit values:

PM_GRAPHICAL_INPUT
Input device for graphical output, such as the keyboard and the mouse.

PM_GRAPHICAL_OUTPUT
Output device for graphical input, such as graphics.

PM_AUDIO_INPUT
Input device for audio output device.

PM_AUDIO_OUTPUT
Output device for audio sound.

PM_RING_RESUME_SUPPORT
Device that supports the feature of ringing resume.

PM_REMOTE_TERMINAL
Asynchronous terminal, such as TTY, has this attribute.

The PM core checks the `activity` fields of all devices that have attribute bits set to determine when the system is idle.

The `attribute` field is set to the logical OR of all relevant attributes.

`pm_handle.next1`

The `next1` field is used by the PM core and is not modified by the device driver.

pm_handle.next2
: The next2 field is used by the PM core and is not modified by the device driver.

pm_handle.device_idle_time1
: The device_idle_time1 field is used only by graphics device drivers. This field indicates the period of device inactivity before the PM core requests the graphics device to go to PM_DEVICE_DPMS_STANDBY mode. Device drivers, other than graphics device drivers, should set this field to 0.

pm_handle.device_idle_time2
: The device_idle_time2 field is used only by graphics device drivers. This field indicates the period of device inactivity before the PM core requests the graphics device to go to PM_DEVICE_DPMS_SUSPEND mode. Device drivers, other than graphics device drivers, should set this field to 0.

pm_handle.device_logical_name
: The device_logical_name field is set to a pointer that points to the device's logical name string. For example, a disk driver allocates a string area that contains a corresponding disk name, such as hdisk0 or hdisk1, and sets a pointer that points the string area to the device_logical_name field.

pm_handle.reserve
: The reserve field is reserved and is set to 0.

pm_handle.pm_version
: The pm_version field indicates the supported level of the power management implementation. This field is initialized to 0x0100 if the driver is compliant with this documentation.

pm_handle.extension
: The extension field is reserved for future expansion. The pm_version field directs future usability of this field. The extension field is initialized to 0.

pm_handle Fields Retrieved from ODM

The following data of each PM-aware device driver is stored in the ODM PdAt object class and retrieved at device configuration time so that the corresponding PM-aware device driver can set them in the **pm_handle** structure:

```
device attribute                (attribute name: pm_dev_att)

default device idle time        (attribute name: pm_dev_itime)

default device standby time     (attribute name: pm_dev_stime)

        Actual instance of hard disk
                        PdAt:
                                uniquetype="disk/scsi/540mb2"
                                attribute="pm_dev_itime"
                                deflt="300"
                                values="0-7200,1"
                                width=""
                                type="R"
                                generic=""
                                rp="nr"
                                nls_index=0
```

pm_planar_control

The **pm_planar_control** kernel service allows a device driver to request planar-level power management control of its device regardless of the current platform. Some devices are power-managed through external logic to a device slot. The **pm_planar_control** kernel service abstracts this functionality from the device driver.

The **pm_planar_control** kernel service can be called from either the process or interrupt environment.

All devices should include calls to **pm_planar_control**, even if the device has built-in power management features. This provides a common programming interface and ensures that the **pm_planar_control** kernel service has a chance to manage planar power control, where available.

The following planar device IDs are used to control device power through the PM planar control kernel service. These values are defined in the **sys/pmdev.h** header file.

```
Display indicator
  LCD                   PMDEV_LCD                   0x00010000
  CRT                   PMDEV_CRT                   0X00010100
                                (PM_PLANAR_LOWPOWER1=DPMS standby mode)
                                (PM_PLANAR_LOWPOWER2=DPMS suspend mode)
                                (PM_PLANAR_OFF      =DPMS off mode)

Video controller
  graphic controller    PMDEV_GCC                   0X00020000
  DAC                   PMDEV_DAC                   0X00020100
  VRAM                  PMDEV_VRAM                  0X00020200

multi media
  video capture         PMDEV_VCAP                  0X00030000
  playback              PMDEV_VPLAY                 0X00030100
  CCD camera            PMDEV_CAMERA                0X00030200
  audio                 PMDEV_AUDIO                 0X00030300
  audio mute            PMDEV_AUDIO_EXT_MUTE        0X00030310

graphical input
  internal keyboard     PMDEV_INTKBD                0X00040000
  external keyboard     PMDEV_EXTKBD                0x00040100
  internal mouse        PMDEV_INTMOUSE              0X00040200
  external mouse        PMDEV_EXTMOUSE              0X00040300

communication device
  serial1               PMDEV_SERIAL1               0x00050001
  serial2               PMDEV_SERIAL2               0x00050002
  parallel              PMDEV_PARALLEL              0x00050100

CPU local bus
  CPU                   PMDEV_CPU                   0x00090000(LOWPOWER1=doze)
  L2 cache              PMDEV_L2                    0x00090100(OFF=cache flush)
  others                PMDEV_LOCALn(n=2-f)         0x00090200-0x00090f00

extended bus slot
  PCMCIA(slot 0-f)      PMDEV_PCMCIANn(n=0-f)       0x000c00N0-0x000c00Nf
                        N=0-f(bus#)
  PCI(slot 0-f)         PMDEV_PCINNDD               0x000dNNDD
                        NN=0-ff(bus #)
                        DD=DevFunc# (device number<<3|function number)
  ISA(slot 0-f)         PMDEV_ISANn(n=0-f)          0x000e00N0-0x000e00Nf
                        N=0-f(bus #)

unknown device
  SCSI device           PMDEV_UNKNOWN_SCSI          0X00100iii
                        iii=((SCSI ID<<6) | LUN)
```

```
  IDE                   PMDEV_UNKNOWN_IDE           0X0011000n
                        n=device number
  Others                PMDEV_UNKNOWN_OTHER         0X00120000

internal device
  SCSI drive            PMDEV_INTERNAL_SCSI         0X0018xiii
                        x=power connector number (0-f)
                        iii=((SCSI ID<<6) | LUN)
  IDE                   PMDEV_INTERNAL_IDE          0X0019x00n
                        x=power connector number (0-f)
                        n=device number
  Others                PMDEV_INTERNAL_OTHER        0X001ax000
                        x=power connector number (0-f)

Others
  FDC                   PMDEV_FDC                   0X00060000
```

The planar device ID of the extended bus slot and the internal and unknown device planar device ID are variable. The device driver must OR in the variable values with the base planar device ID. In the case of the extended bus slot, the device driver builds the entire planar device ID. For internal and unknown devices, there must be a device attribute in ODM (**pm_devid**) that indicates the base device ID. The default value is either PMDEV_UNKNOWN_SCSI, PMDEV_UNKNOWN_IDE, or PMDEV_UNKNOWN_OTHER, depending on the device type. The driver uses this value and ORs in the variable values (such as SCSI ID or lun). The PMDEV_INTERNAL_SCSI, PMDEV_INTERNAL_IDE, or PMDEV_INTERNAL_OTHER base devid provides an interface for the user to indicate whether a certain device is internal and what its power connector ID is. Since this is passed by the driver to the **pm_planar_control**, this allows the machine-dependent planar_control function to manage the power to these connectors if that functionality is provided on that platform.

An example of the PM planar control's support for further power control functions, in addition to the function that the device driver directly handles, is the CD-ROM device driver. In addition to spinning down the motor through a conventional SCSI command, the CD-ROM drive can be turned on or off using the PM planar control if the drive is attached internally in the machine. However, the CD-ROM driver does not actually consider the difference between an internal device and an external (unknown) one. A special attribute that indicates "internal" or "unknown" is set. The CD-ROM driver uses this information as the bit 12 of **planar_devid**. As a result, the **planar_devid** is different between an internal device and an unknown device. For an internal device, the power connector number can be input into SMIT based on user information, in addition to the internal or unknown attribute.

pm_register_planar_control_handle

The **pm_register_planar_control_handle** kernel service allows the actual planar control function of the PM core to be extended. The **pm_planar_control** kernel service takes the devid passed to it and looks for a registered **planar_control_handle** function for that devid and, if it exists, calls it.

For example, on a platform with PCMCIA, the PCMCIA card/socket services call **pm_register_planar_control_handle( )** to register planar control functions for each PCMCIA slot. Then, when a PCMCIA device is attached and calls the **pm_planar_control** kernel service, the appropriate PCMCIA **planar_control_handle( )** function is called.

General Model of PM-Aware Device Driver

This model of a PM-aware device driver contains the following sample code:

- **Device_PM_Handler( )**
- **Device_external_interrupt_handler( )**
- **StartIO( )**

Device_Pm_Handler()

```
/*
_____________________________________________________________________
 A NEW handler with Power Management entry
_____________________________________________________________________
*/
device_pm_handler (caddr_t private, int ctrl)                    /* New routine for PM */
{
   switch(ctrl) {
   case suspend:
   case hibernation:
            if (outstanding condition in waiting something to complete) {
                     if (It takes more than 1 second even in normal case) {
                              return (PM_ERROR);
                     } else {
                              pm_pending = TRUE;
                              e_sleep(&suspend_hib_pending, ...);
                              /* ###  pm_pending ###*/
                              /* pm_pending flag is used to wait the current outstanding IO
                              operation to complete. This flags gets the interrupt handler
                              to call the e_wakeup corresponding to the above e_sleep when
                              the waited event completes. */
                     }
            }
            if (ctrl == suspend) {
                     mode = suspend;
                     pm_planar_control(planar_devid, PM_PLANAR_OFF);

            } else {
                     mode = hibernation;
            }
            suspend_hibernation_job();
            if (having async external int || needs to turn the device on immediately after resume completion)
                     activity flag = –1;
            break;

   case device_idle:
   case PM_enable:
            if (ctrl == PM_enable && mode == device_idle) {
                     if (planar_off was applied for device_idle) {
                              pm_planar_control(planar_devid, PM_PLANAR_ON);
                              retrieve_device(from_off_to_enable);
                     } else if (planar_lowpower was applied for device_idle) {
                              pm_planar_control(planar_devid, PM_PLANAR_ON);
                              retrieve_device(from_idle_to_enable);
                     }
                     if (device specific internal low power mode was applied) {
                              retrieve the device to normal mode (e.g. restart disk motor);
                     }
                     mode = PM_enable;
            }
            if (ctrl == device_idle && mode == PM_enable) {
                     if (planar_off is intended to be applied for device_idle mode) {
```

```
                        Save the current device context if needed;
                        pm_planar_control(planar_devid, PM_PLANAR_OFF);
                } else if (planar_lowpower1 is intended to be applied for device_idle mode) {
                        Save the current device context if needed;
                        pm_planar_control(planar_devid,\
                                                PM_PLANAR_LOWPOWER1);
                }
                if (device specific low power mode is intended to be applied) {
                        Get the device to enter the low power mode (e.g. stop disk motor);
                }
                mode = device_idle;
        }
        if (mode == suspend || mode == hibernation) {
                if (block_for_pm == TRUE) {
                        block_for_pm = FALSE;
                        e_wakeup(&pm_block);
                        /*  ### block_for_pm ### */
                        /*  block_for_pm  flag is used to block the further device request
                        which  this driver receives after it accepts the request of PM
                        device mode transition with success.  Here, the blocking is
                        terminated because the resume occurs. */
                }
                if (ctrl == device_idle) {
                        if (planar_off is NOT applied for device_idle) {
                                pm_planar_control(planar_devid,\
                                        PM_PLANAR_ON);
                                retrieve_device(from_off_to_idle);
                                pm_planar_control(planar_devid,\
                                        PM_PLANAR_LOWPOWER1);
                                /* The device has been in PM_PLANAR_OFF.
                                To move on to LOWPOWER1, it needs to
                                be temporarily turned on(PLANAR_ON)
                                once.  Thus, here is PM_PLANAR_ON
                                before PM_PLANAR_LOWPOWER1. */
                        }
                        mode = device_idle;
                } else {
                        pm_planar_control(planar_devid,PM_PLANAR_ON);
                        retrieve_device(from_off_to_enable);
                        mode = PM_enable;
                }
        }
        break;

case full_on:
        if (mode != PM_enable)
                return (PM_ERROR);
        Disable device_level power management;
        mode = full_on;
        break;

case device_DPMS_standby:
        if (!(attribute & PM_GRAPHICAL_OUTPUT))
                return (PM_ERROR);
        pm_planar_control(planar_devid,PM_PLANAR_LOWPOWER1);
        Apply DPMS standby operation to graphics sub–system;
        mode = device_DPMS_standby;
        break;

case device_DPMS_suspend:
        if (!(attribute & PM_GRAPHICAL_OUTPUT))
                return (PM_ERROR);
        pm_planar_control(planar_devid,PM_PLANAR_LOWPOWER2);
```

```
            Apply DPMS suspend operation to graphics sub–system;
            mode = device_DPMS_suspend;
            break;

    case PM_page_freeze_notice:
            if (all memory for PM related code/data are already pinned)
                    return (PM_SUCCESS);
            Pin all code/data related to power management operations;
            if (additional data area is needed for power management)
                    Get the area through ”xmalloc” and pin it;
                    if (not enough memory can’t be obtain)
                            return(PM_ERROR)
            break;

    case PM_page_unfreeze_notice:
            if (all memory for PM related code/data are already pinned)
                    Unpin all code/data related to power management operations;
                    return(PM_SUCCESS);
            if (additional data area was obtained  for power management)
                    Release the area;
            break;

    case PM_MDM_ring_resume_notice:
            if (!(attribute & PM_RING_RESUME_SUPPORT))
                    return (PM_ERROR);
            Enable ring resume feature;
            break;
    }
     return (PM_SUCCESS);
}
```

Device_external_interrupt_handler()

```
/*
_____________________________________________________________________
  Interrupt handler MODIFIED for Power Management
_____________________________________________________________________
*/
Device_external_interrupt_handler()                     /* Modified routine for PM */
{
    appropriate_interrupt_routine();
    Set activity flag to either 1 or –1;

    if (This int causes the current outstanding IO request to complete) {

            if (pm_pending == TRUE) { /* Check if pm_handler is now waiting for the completion of the
            outstanding IO operation to successfully accept the request of
            suspend or hibernation from PM core.  */

                    if (it is in a boundary of the processing) {
                            pm_pending = FALSE;
                            e_wakeup(&suspend_hib_pending);
                    }
            }
    }
    return (INTR_SUCC);
}
```

StartIO()

```
/*
_____________________________________________________________________

  Device handler MODIFIED for Power Management
_____________________________________________________________________
*/
startIO()                                          /* Modified routine for PM */
{
   switch(mode) {
   case suspend:
   case hibernation:
            block_for_pm = TRUE;
            e_sleep(&pm_block,...);
            /* Since the request of suspend or hibernation has already
            been accepted with success, any further device request
            must not be processed until resume notice such as device_idle
            request or PM_enable request.  This pm_block is checked
            at the routine of resume sequence described above and then
            the above e_sleep is unblocked.  */

            if (request needs to access the device directly)
                     Set activity flag to either 1 or –1;
            break;

   case device_idle:
            if (request needs to access the device directly) {
                     Set activity flag to either 1 or –1;
                     pm_planar_control(planar_devid,PM_PLANAR_ON);
                     retrieve_device_state(from_idle_to_enable);
            }
            break;

   case PM_enable:
            if (request needs to access the device directly) {
                     Set activity flag to either 1 or –1;
                     if (device level power management is applied)
                              Set the device/a part of the sub–system to normal state;
            }
            break;

   case full_on:
            do nothing;
            break;
   }
   appropriate_startIO_routine();
}
```

PM-Aware PCMCIA Device Drivers

All PCMCIA clients are required to be PM-aware. The following are some PCMCIA-unique requirements for PM implementation:

- At the beginning of the resuming routine from the suspend and hibernation states, all clients must check tuples in the Card Information Structure (CIS) to determine whether the card in the slot is the expected one. Even though configuration has changed during the suspend and hibernation states, the CSE_CARD_REMOVAL and CSE_CARD_INSERTION events are not issued from card services.
- All resources are not required to be released to card services when the clients enter the suspend and hibernation states. However, if the card is removed during those states, the clients should release all remaining resources to card services in the resuming process. Since there is a possibility that a PC card has been removed and reinserted in the same socket, PCMCIA clients might need to restore the PC card's configuration registers when resuming from the suspend state, even though an expected card is in the previous socket.
- If the clients expect ringing resume in the suspend state, the client must set the configuration registers (for example, Card Configuration and Status Register (CCSR)) to route the ring indication to the PC Card Interface Controller (PCIC) and keep the power on.
- While the clients are in the suspend and hibernation states, they should reject and return any callback events from card services. Therefore, the clients should not make any requests that cause callback events to other clients. The RequestExclusive, ReleaseExclusive, and GetClientInfo card services functions are restricted in the routine of processing the entering of suspend, hibernation, and resume.

  In addition, if a client calls the ResetCard function, the client should be using a socket exclusively by calling the RequestExclusive function in advance.
- The RegisterClient, DeregisterClient, RequestSocketMask, and ReleaseSocketMask card services functions should not be called from the client's **pm_handler** routine, but from the **config** or **unconfig** routine.

  Note: The **pm_planar_control** routine for each socket that is provided by card services supports PM_PLANAR_QUERY, PM_PLANAR_ON, PM_PLANAR_OFF, and PM_PLANAR_CURRENT_LEVEL.

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Vos remarques sur ce document / Technical publication remark form

Titre / Title : Bull DPX/20 Writing a Device Driver

Nº Reférence / Reference Nº : 86 A2 29WG 04

Daté / Dated : November 1995

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : _____ Date : _____

SOCIETE / COMPANY : _____

ADRESSE / ADDRESS : _____

_____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

BULL S.A. CEDOC
Atelier de Reproduction
FRAN–231
331 Avenue Patton BP 428
49005 ANGERS CEDEX
FRANCE

BULL S.A. CEDOC
Atelier de Reproduction
FRAN–231
331 Avenue Patton BP 428
49005 ANGERS CEDEX
FRANCE

ORDER REFERENCE
86 A2 29WG 04

PLACE BAR CODE IN LOWER LEFT CORNER

Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

DPX/20
AIX
Writing a Device Driver

86 A2 29WG 04

DPX/20
AIX
Writing a Device Driver

86 A2 29WG 04

DPX/20
AIX
Writing a Device Driver

86 A2 29WG 04