PROGRAMMING

DEVICE DRIVER PROGRAMMING

UNIX® SVR4.2

Edited by Robert M. Hines and Spence Wilcox

DEVICE DRIVER PROGRAMMING

UNIX SVR4.2

Edited by Robert M. Hines and Spence Wilcox

UNIX
Press

Published by Prentice-Hall, Inc.
A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

IMPORTANT NOTE TO USERS

While every effort has been made to ensure the accuracy and completeness of all information in this document, USL assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors, omissions, or statements result from negligence, accident, or any other cause. USL further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. **USL disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness for a particular purpose.** USL makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting of any license to make, use or sell equipment constructed in accordance with this description.

USL reserves the right to make changes to any products herein without further notice.

TRADEMARKS

UNIX is a registered trademark of UNIX System Laboratories, Inc. in the USA and other countries. Intel386 and Intel486 are trademarks of Intel Corp.

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-042623-7

UNIX
PRESS
A Prentice Hall Title

PRENTICE HALL

ORDERING INFORMATION

UNIX® SYSTEM V RELEASE 4.2 DOCUMENTATION

To order single copies of UNIX® SYSTEM V Release 4.2 documentation, please call (515) 284-6761.

ATTENTION DOCUMENTATION MANAGERS AND TRAINING DIRECTORS:
For bulk purchases in excess of 30 copies, please write to:

Corporate Sales Department
PTR Prentice Hall
113 Sylvan Avenue
Englewood Cliffs, N.J. 07632

or

Phone: (201) 592-2863
FAX: (201) 592-2249

ATTENTION GOVERNMENT CUSTOMERS:

For GSA and other pricing information, please call (201) 461-7107.

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Figures and Tables

1 Introduction to Device Drivers

Introduction

This document, *Device Driver Programming*, provides information and procedures for developing, installing, and testing UNIX® System V device drivers. The introductory chapter of this guide is intended primarily for programmers writing device drivers that use the traditional UNIX system block and character driver interfaces. The remaining chapters describe system features and programming procedures used by all driver writers, regardless of the kind of interface used (block, character, STREAMS, or Portable Device Interface).

Since the common material in *Device Driver Programming* does not appear in the other two titles in the device driver programming documentation set— *STREAMS Modules and Drivers* and *Portable Device Interface (PDI)*— readers of the guides for these alternate interfaces should also read this guide.

Contents

This guide contains four chapters:

- Chapter 1, "Introduction to Device Drivers", introduces many of the basic concepts a programmer should understand before attempting to write a UNIX System V device driver.
- Chapter 2, "Loadable Modules", discusses Dynamically Loadable Modules (DLM), a feature that allows you to add a device driver to a running system without rebooting the system or rebuilding the kernel. The first part provides an overview of the DLM feature from the driver writer's perspective. The second part explains how to convert your non-loadable driver to be loadable.
- Chapter 3, "Driver Installation and Tuning", explains how to install and configure device drivers using Installable Driver Tools (also known as idtools) and Driver Software Packages (DSPs). Information on tuning device drivers is also provided.
- Chapter 4, "Driver Testing and Debugging", describes the tools that are available for testing and debugging a device driver, and discusses some of the common errors and some of the symptoms that might identify each.

A glossary of common UNIX system device driver programming terms and abbreviations is also provided.

Changes since Previous Release

Device Driver Programming is a new title that covers many of the topics addressed by the "Device Drivers" chapter of the *Integrated Software Development Guide* (ISDG) in previous releases. For Release 4.2, a few sections that formerly appeared in this ISDG chapter have been updated and reused. However, the majority of the material in *Device Driver Programming* is entirely new material. The most significant technical changes for Release 4.2 are documented in Chapter 2, "Loadable Modules", and Chapter 3, "Driver Installation and Tuning".

References

The following UNIX System V reference manuals are a recommended supplement to this guide:

- *Command Reference* (Section 1)
- *Operating System API Reference* (Sections 2 and 3)
- *Windowing System API Reference* (Section 3 windowing functions)
- *System Files and Devices Reference* (Section 4, 5, and 7)
- *Device Driver Reference* (Sections D1 - D5)

These books contain the manual pages for the various commands, system calls, library functions, file contents, and devices. Within each book, manual pages are grouped numerically by section numbers. Within a section, the pages are sorted alphabetically, without regard to the letter that follows the section number. For example, the manual pages for Sections 3C, 3E, 3I, 3M, 3N, 3S, 3W, and 3X are all sorted together within Section 3 in the *Operating System API Reference*.

Notation Conventions

The following conventions are observed in this guide:

- Computer input and output appear in **`constant width`** type. This includes program code, specific file names and contents, and commands.
- Substitutable values, such as file or device names that you set and variables, appear in *italic* type.

Following is an example demonstrating how both constant width font and italics are used throughout this guide.

`Master` files contain lines of the form:

`$version` *version-number*
`$entry` *entry-point-list*
`$depend` *module-name-list*
`$modtype` *loadable-module-type-name*
module-name prefix characteristics order bmaj cmaj

Chapter Overview

The remainder of this chapter introduces many of the basic concepts a programmer should understand before attempting to write a UNIX System V device driver. The chapter gives an experienced C programmer an overview of how to write a device driver, by showing

- how device drivers resemble and differ from application programs
- different types of device drivers, and what they have in common with each other
- some of the standard driver-to-kernel and driver-to-hardware interface routines, and where to find additional information about these interfaces
- the structures used by the system to provide driver entry points
- methods used to differentiate between devices and subdevices
- an example driver that illustrates the main components of most drivers and what those components typically do
- some guidelines for developing a driver

What Is a Device Driver?

The UNIX operating system kernel can be divided into two parts: the first part manages the file systems and processes, and the second part manages physical devices, such as terminals, disks, tape drives, and network media. To simplify the terminology, this chapter refers to the first part as "the kernel" (although strictly speaking, drivers are part of the kernel too), and refers to the second part, which contains the drivers, as "the I/O subsystem."

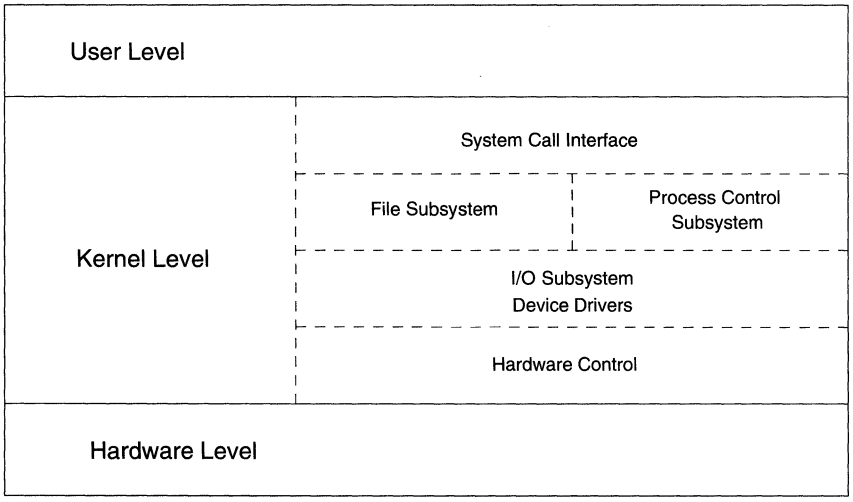
Associated with each physical device is a piece of code, called a device driver, which manages the device hardware. The device driver brings the device into and out of service, sets hardware parameters in the device, transmits data from the kernel to the device, receives data from the device and passes it back to the kernel, and handles device errors.

To most application programmers using UNIX System V, a device driver is simply part of the operating system. The application programmer is usually concerned only with opening and closing files and reading and writing data. These functions are accomplished through standard system calls from a high-level language. The system call gives the application program access to the kernel, which identifies the device containing the file and the type of I/O request. The kernel then executes the device driver routine provided to perform that function.

Device drivers isolate low-level, device-specific details from the system calls, which can remain general and uncomplicated. Because there are so many details for each device, it is impractical to design the kernel to handle all possible devices. Instead, a device driver is included for each configured device. When a new device or capability is added to the system, a new driver must be installed.

Figure 1-1 shows how a driver provides a link between the user level and the hardware level. By issuing system calls from the user level, a program accesses the file and process control subsystems, which, in turn, access the device driver. The driver provides and manages a path for the data to or from the hardware device, and services interrupts issued by the device's controller.

Figure 1-1: Driver Placement in the Kernel



Every device on a UNIX system looks like a file. In fact, the user-level interface to the device is called a "special file." The device special files reside in the **/dev** directory, and a simple **ls** will tell you quite a bit about the device. For example, the command **ls -l /dev/lp** yields the following information

```
crw-rw-rw-   1 root     root      4,  0 Jul 26 12:45 /dev/lp
```

This says that the **lp** (line printer) is a character type device (the first letter of the file mode field is **c**) and that major number **4**, minor number **0** is assigned to the device. More will be said about device types, and both major and minor numbers, later in this chapter.

Application Programs Versus Drivers

Most applications and drivers are written in the C programming language. However, there are some major differences between writing a device driver and writing a program designed to execute at the user level. This section reviews some of those differences and introduces some of the system facilities used in driver development.

Structure

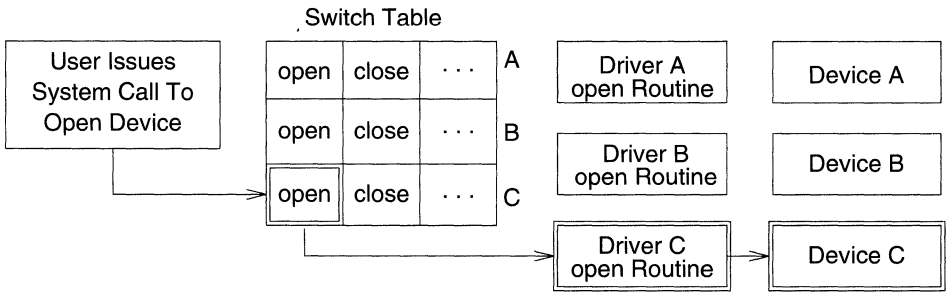
The most striking difference between a driver and a user-level program is its structure. An application program is compiled into a single, executable image whose top-level structure is determined by a **`main`** routine. Subordinate routines are called in the sequence controlled by the **`main`** routine.

A driver, on the other hand, has no **`main`** routine. Rather, it is a collection of routines installed as part of the kernel. But if there is no **`main`** routine to impose structure, how do the driver's routines get called and executed?

Driver routines are called, as needed, in response to system calls or other requirements. System data structures, called switch tables, contain the starting addresses for the principal routines included in all drivers. In a switch table, there is one row for each driver, and one column for each standard routine. The standard routines are called entry-point routines, referring to the memory address where the routine is entered. The kernel translates the arguments of the system call into a value used as an index into the switch table.

For example, when a user process issues a system call to open a file on a device that has a driver, the request is directed to the switch table entry for an open of the device driver containing the file (see Figure 1-2). This routine is then executed, giving the process access to the file.

Figure 1-2: How Driver Routines Are Called



Parallel Execution

When an application program is running, the statements making up the program are executed one at a time, in sequential order. Program control structures (loops and branches) repeat statements and may branch to alternative sections of code, but the important point is that at any given instant only one statement and one routine is being executed. This is true even of different instances of a program being run by two users at the same time (for example, a text editor). As each process is assigned a scheduled slice of CPU time, the statements are executed in the order maintained for that invocation of the program.

Drivers, however, are part of the kernel and must be ready to run as needed at the request of many processes. A driver may receive a request to write data to a disk while waiting for a previous request to complete. The driver code must be designed specifically to respond to numerous requests without being able to create a separate executable image for each request (as a text editor does). The driver does not create a new version of itself (and its data structures) for each process, so it must anticipate and handle contention problems resulting from overlapping I/O requests.

Interrupts

For the most part, the real work of a device driver is moving data between user address space and a hardware device, such as a disk drive or a terminal. Because devices are typically very slow compared to the CPU, the data transfer may take a relatively long time. To overcome this, the driver normally suspends execution of the process until the transfer is complete, freeing the CPU to attend to other processes. Then, when the data transfer is complete, the device sends an interrupt, which tells the original process that it may resume execution.

The processing needed to handle hardware interrupts is another of the major differences between drivers and application programs.

Driver As Part of the Kernel

Application programs, executing at the user level, are limited in the ways they can have an adverse impact upon the system. Performance and efficiency considerations are mostly confined to the program itself. An application program can consume excessive disk space, but it cannot raise its own priority level to use excessive amounts of processing time, nor does it have access to sensitive areas of the kernel or other processes.

But drivers can and do have much greater impact on the kernel. Inefficient driver code can severely degrade overall performance, and driver errors can corrupt or bring down the system. For this reason, testing and debugging driver code is particularly challenging, and must be done carefully. Chapter 4 discusses the facilities available for finding driver errors, as well as some of the special problems that are encountered when testing driver code.

Also, while an application program writer is free (within reasonable limits) to declare and use data structures and to use system services, a driver writer is constrained in several ways.

- Kernel functions called by the driver generally do not verify the validity of passed arguments. Therefore, it is the responsibility of the driver developer to check the validity of arguments before passing them to kernel functions.
- A number of header files, used to declare data types, initialize constants, and define system structures, must be included in the driver source code. The exact list of header files varies from driver to driver; some of the commonly-used header files are described later in this chapter.

- Various structure members and device registers must be read or written, and usually some system buffering structure must be used. Many of the functions defined in the UNIX system Device Driver Interface/Driver-Kernel Interface (DDI/DKI) are designed to be used with these structures. These structures are explained in Section D4, "Kernel Data Structures", of the DDI/DKI portion of the *Device Driver Reference*.
- Drivers have no access to standard C library routines; however, the routines included in the DDI/DKI represent a kind of library and provide some functions similar to those found in the standard C library. On the other hand, the DDI/DKI also provides many functions that are unlike standard C library functions. See Section D3, "Kernel Utility Routines", of the DDI/DKI portion of the *Device Driver Reference* for complete explanations of the driver interface routines.

NOTE

Some of the DDI/DKI functions [such as `rmalloc`(D3DK)] are similar to standard library functions [in this case, `malloc`(3C)], but have different arguments. Serious errors could result if the driver writer does not pay attention to such differences.

- Drivers are invoked by the kernel using a set of system tables and the standard C function-calling mechanism. Every member of one of these tables is a structure containing pointers to the driver's entry point routines. The entry point routines make the connection between the calling process and the device driver. The entry points, in turn, call the driver functions to service the caller's requests. See Section D2, "Driver Entry Point Routines", of the DDI/DKI portion of the *Device Driver Reference* for complete explanations of the driver entry point routines.
- Drivers cannot use floating point arithmetic.

Types of Devices

So far, interactive terminals and disk drives have been mentioned as two kinds of devices that need drivers. These two kinds of devices use very different types of drivers. On any UNIX system processor, there are two kinds of devices: hardware devices and software, or pseudo-devices.

Hardware Devices

Hardware devices include familiar peripherals such as disk drives, tape drives, printers, ASCII terminals, and graphics terminals. The list could also include optical scanners, analog-to-digital converters, robotic devices, and networks. But, in reality, a driver never talks to the actual piece of hardware, but to its controller board. From the point of view of the driver, the device is usually a controller.

In some cases, a controller may have only one device connected to it. More often, several devices are connected to a single board (for example, eight terminals could be connected to a terminal controller). A single driver is used to control that board and all similar terminal controllers configured into the system.

Software Devices

The "device" driven by a software driver is usually a portion of memory and is sometimes called a pseudo-device. The driver's function may be to provide access to system structures unavailable at the user level.

For example, a software device might be a RAM disk, which provides very fast access to files by using a part of memory for mass storage. A RAM disk driver is, in many ways, similar to a driver for an actual disk drive, but does not have to handle the complications introduced by actual hardware. The sample driver (shown later in this chapter) is a RAM disk driver.

Types of Device Driver Interfaces

A device driver interface is the set of structures, routines, and optional functions used to implement a device driver. UNIX System V Release 4.2 provides three device driver interfaces, all of which are based upon a single specification, the Device Driver Interface/Driver Kernel Interface (DDI/DKI).

Block and Character Interface

Block and character are the two traditional UNIX system device driver interfaces, and they correspond to the two basic ways drivers move data. Block drivers, using the system buffer cache, are normally written for random-access devices such as disk drives and any mass storage devices capable of handling data in independently addressable blocks. Character drivers, the typical choice for interactive terminals, are normally written for devices that send and receive information one character at a time.

It is the individual device and goal of the implementation, not the device type, that determines whether a driver should be the block or character type. For example, one driver developer may want to implement a driver for a 9-track tape controller such that file system images on the tape would be mountable, even though performance of the tape controller for random block accesses would not be good. Another driver developer may choose to view the tape as a device that can only be used for sequential storage and retrieval of data, and hence write only a character driver.

Furthermore, one device may have more than one interface. A disk drive, for example, may have both a block and character interface.

The manual pages for the block and character interfaces can be found in the DDI/DKI sections of the *Device Driver Reference*.

STREAMS Interface

In some early UNIX system releases, the increasing number of network drivers demonstrated one of the major weaknesses of the block and character interface: its inability to divide a network's protocols into layered modules. The solution, first introduced in UNIX System V Release 3, is called the STREAMS interface.

A stream is a structure made up of linked modules, each of which processes the transmitted information and passes it to the next module. One of these queues of modules connects the user process to the device, and the other provides a data path from the device to the process.

The layered structure allows protocols to be stacked and also increases the flexibility of the interface, making it more likely that modules can be used by more than one driver.

In UNIX System V Release 4 the character-based TTY subsystem was reimplemented using STREAMS, and the character-based TTY subsystem is now supported only for compatibility (its interfaces are not part of the DDI/DKI specification).

For information about STREAMS drivers, refer to the guide *STREAMS Modules and Drivers*.

The manual pages for the STREAMS interface can be found in the DDI/DKI sections of the *Device Driver Reference*.

Portable Device Interface (PDI)

With Release 4.2, UNIX System V provides an architecture for the development of block-oriented device drivers called the Portable Device Interface (PDI). The PDI emphasizes the separation of hardware-dependent and hardware-independent parts of drivers. It consists of a collection of driver routines, kernel functions, and data structures that complement, and are based upon, the DDI/DKI interfaces. Included in the PDI is an interface (called SCSI Driver Interface or SDI) for writing target drivers to access Small Computer System Interface (SCSI) devices.

For information about the PDI, refer to the guide *Portable Device Interface (PDI)*.

The manual pages for the Portable Device Interface can be found in the PDI sections of the *Device Driver Reference*.

Major and Minor Numbers

Before the operating system can provide access to a device, the driver must be installed and a special device file must be created in **/dev**.

The special device file contains the major and minor device numbers.

Major Numbers

The major number identifies the device class or group, such as a controller for several terminals (for example, it tells the kernel which driver's **open** routine to call). The major number is assigned, sequentially, to each device driver by the Installable Driver Tools (idtools) during driver installation. Assignment is made by creating an entry in one of the driver's system configuration files, the **Master** file, which is described later in this chapter.

Character major numbers and block major numbers are assigned separately for devices that are exclusively block or character. This means that two separate special files for two different device drivers may appear to have the same number assigned to them. A device that supports both block and character access (for example, the floppy driver), may have different major numbers for the character and block device files.

Minor Numbers

The minor number identifies a specific device, such as a single terminal. Minor numbers are assigned to special files by the driver writer in another system configuration file called the **Node** file (see the **Node**(4) manual page).

Minor numbers are typically used to distinguish subdevices, but they can also be used to convey other information. For example, consider a floppy disk controller that can read and write data from floppies in several formats, and can also manage two floppy drives. When a special file associated with the floppy driver is opened, the minor number used to open the file must tell the floppy driver both which drive to access, and what format to assume for the I/O operation. In this particular case, the least significant bit of the minor number could be used to identify the drive, and the remaining bits used to indicate the format.

Driver Entry Points and Kernel Utilities

Entry Points

Drivers are accessed in three ways

- through system initialization
- through system calls from user programs
- through device interrupts

When the system is initialized, several tables are created so that the system can activate the correct driver routine. Because the system uses these tables to determine the appropriate driver routines to enter, the routines themselves are sometimes referred to as driver entry points.

Each table is associated with a specific set of entry-point routines. Initialization tables are associated with either **init**(D2D) or **start**(D2DK) routines. System calls use a pair of switch tables whose entry points include **open**(D2DK), **close**(D2DK), **read**(D2DK), **write**(D2DK), and **ioctl**(D2DK) routines (for character drivers), and **open**, **close**, and **strategy**(D2DK) routines (for block drivers). STREAMS drivers are entered initially through the character switch table, but their **open**, **close**, **put**(D2DK), and **srv**(D2DK) routines are accessed indirectly through a chain of pointers to other structures. Device interrupts are associated with their appropriate interrupt handling routine through an interrupt vector table. The entry point is the **intr**(D2D) routine.

This section discusses these system tables and their associated entry points in greater detail.

Initialization Entry Points

Boths kinds of driver initialization routines (**init** and **start**) are executed during system initialization, in a different order each time the system is configured. The system uses the routines and information from the driver's configuration files to initialize the drivers. Information such as the major/minor numbers, important when accessing driver switch table entry points, is not used to initialize a driver. The system does not differentiate between character- and block-access drivers when running the initialization routines.

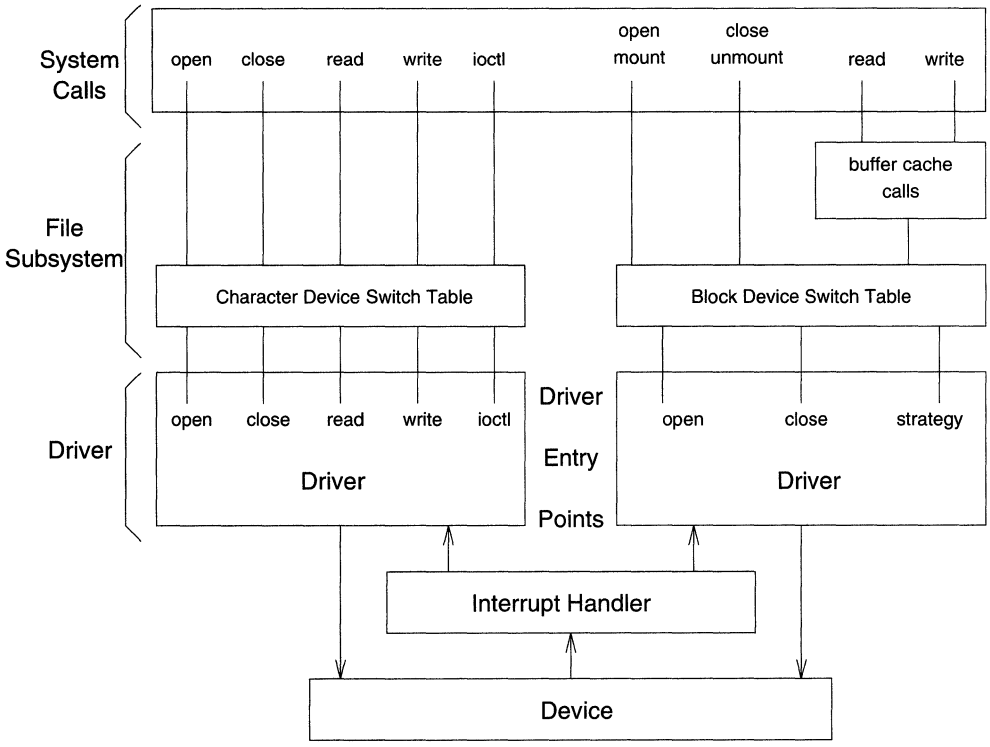
The system initialization program first creates two internal tables, **io_init** and **io_start**, which it uses to list the routines that must be executed. After the system is initialized, the **io_init** and **io_start** tables are not accessed again. Not

all drivers need initialization routines. A driver that does not have an **init** or **start** routine has no entry in the **io_init** or **io_start** table.

Switch Table Entry Points

Two operating system switch tables, **cdevsw** and **bdevsw**, hold the entry-point routines for character and block drivers, respectively. These routines are activated by I/O system calls (Figure 1-3).

Figure 1-3: Switch Table Entry Points and System Calls



The process of calling the appropriate driver routine can be summarized as follows

1. The I/O system call (**open** and **read**, for example) is directed to a special device file.

2. The special device file includes the major number for the driver that controls the device.
3. If the special device file is for block access, the system uses the major number as an index into the **bdevsw** table to find the appropriate routine.

   For character access, the operating system looks in the **cdevsw** table, using the same method.
4. The operating system calls the appropriate routine.

Whenever the character (or block) entry points are being used, the other entry points are inaccessible. When the driver does a character-access **read** or **write** operation on a device that supports both block and character access, it calls the **strategy** routine. The driver calls the **strategy** routine, however, as a subordinate routine to **read** or **write**, not as the **bdevsw** entry point.

STREAMS drivers, although they use the **cdevsw** table, do not use the usual entry points. Instead, a STREAMS driver is recognized by a non-null value in the **d_str** field of **cdevsw**, which is a pointer to a **streamtab**(D4DK) structure. The **streamtab** structure contains pointers to other structures which eventually point to STREAMS entry points.

Although the **bdevsw** and **cdevsw** tables have places for all possible driver routines, not all routines are appropriate for all devices. For instance, a printer driver does not need a **read** routine. The operating system provides place holders in the switch tables for routines that are not included in the driver. The place holder routines are named **nulldev** and **nodev**. **nulldev** is an empty routine that is called when the routine it represents is not needed (for example, a **halt** routine for a printer driver would not be needed because it would have no work to do). **nodev** is a routine that returns an error code when the routine it represents is called (for example, a **read** routine for a printer driver would create an error condition).

Interrupt Entry Points

The operating system must handle many kinds of system interrupts (such as clock and software interrupts), system exceptions (such as page faults), and interrupts from peripheral devices controlled by drivers. Interrupts cause the processor to stop its current process and to immediately begin to service the interrupt. Peripheral devices generate interrupts when an I/O transfer encounters an error or completes successfully.

When an interrupt is received from a hardware device, the kernel determines the interrupt vector number of the device and passes control to the appropriate driver's interrupt handling routine(s). It does this by accessing the interrupt vector table, populated during system initialization. The interrupt handler must

identify the reason for the interrupt (device connect, write acknowledge, data available) and set or clear device state bits as appropriate. It can also awaken processes that are sleeping, waiting for an event corresponding to the interrupt.

Kernel Utility Routines

UNIX system device drivers call kernel utility routines to perform system-level functions. The following sections describe some of the routines typically used in the development of device drivers.

Note that in many cases, routines can be classified under several categories. For example, **biodone** wakes up processes and releases buffers when I/O is complete, therefore falling into the "Event Synchronization Routine," "Buffer Usage Routine," and "I/O Control Routine" categories.

Buffer Usage Routines

A feature common to most drivers is the use of buffers, which are used for handling data. Drivers can use three types of buffers:

KMA buffers
: Kernel Memory Allocator (KMA) buffers are "borrowed" by the driver from a common memory pool used by all parts of the kernel. All types of drivers may use them. When drivers allocate their own data areas or independent buffer pools, this increases the size of the driver, and thus the size of the kernel.

STREAMS message buffers
: STREAMS messages are for use by drivers written to use the STREAMS interface. They are allocated for the driver through the kernel utilities, so the driver need not allocate a pool for its own messages.

System buffers
: System buffers are the size of a file system block, the size of which depends on the type of file system and can vary from 1K to 16K. This buffer pool primarily supports disk I/O operations.

  UNIX System V provides a set of buffers that are normally used for file system I/O, but they can be "borrowed" by drivers if the driver includes the header file **sys/buf.h** and the buffer size if 1024 bytes.

Drivers should be written with the finite nature of the machine in mind; high buffer use by a driver can reduce the performance of other drivers or require more memory to be devoted to buffers. When more memory or space is allocated to buffers, the memory or space available for user processes is correspondingly decreased.

Following are some common buffer usage routines. For information about STREAMS buffers, however, refer to *STREAMS Modules and Drivers*.

Table 1-1: Buffer Usage Routines

Call	Description
`brelse`	Release buffer
`clrbuf`	Clear buffer contents
`freerbuf`	Release buffer from `getrbuf`
`geteblk`	Allocate 1024 byte buffer, return pointer
`geterror`	Return buffer error number
`getrbuf`	Allocate buffer header only
`kmem_alloc`	Allocate space from kernel memory
`kmem_free`	Free allocated kernel memory
`ngeteblk`	Allocate *n*-byte buffer, return pointer

For more information about these and other calls and functions, refer to the DDI/DKI portion of the *Device Driver Reference*.

Data Transfer Routines

Whenever a user program issues a `read`(2) or `write`(2) system call, the operation interacts with data storage areas in the user data space. The driver then moves the data between user space and the device in one of three ways

- directly between user space and the device
- indirectly using local data space in the driver
- indirectly using buffers in kernel memory

Choosing the appropriate data transfer method for your driver depends on the type of the device the driver is supporting, how much intelligence the device supports, and the system utilities that will access it. Many transfers of data between user space and the device require an intermediate transfer of the data into kernel memory.

Table 1-2: Data Transfer Routines

Call	Description
`bcopy`	Copy data from one place to another within kernel address space
`clrbuf`	Clear data in buffer
`copyin`	Copy data from user space to buffer
`copyout`	Copy data from buffer to user space
`rmalloc`	Allocate space from a memory map
`rminit`	Initialize a private memory map
`rmsetwant`	Wait for a free buffer
`rmfree`	Release map entries

For more information about these and other calls and functions, refer to the DDI/DKI portion of the *Device Driver Reference*.

Event Synchronization Routines

An important aspect of driver development concerns how drivers wait for and respond to certain hardware or software events. Driver functions used to suspend the execution of the current process are called under the following circumstances:

- waiting for a hardware action to be accomplished (such as transferring data between a computer and a disk driver)
- waiting for a software action to occur (such as a buffer to be freed for use)
- waiting in a stopwatch mode until a specified number of time units have elapsed

Table 1-3: Event Synchronization Routines

Call	Description
`biodone`	Block I/O wake up
`biowait`	Block I/O sleep
`delay`	Stop current process for a specified time period
`sleep`	Suspend execution until a particular event occurs
`spl`	Set priority level
`timeout`	Timeout function
`untimeout`	Cancel timeout count
`wakeup`	Resume suspended execution

For more information about these and other calls and functions, refer to the DDI/DKI portion of the *Device Driver Reference*.

Interrupt Handling Routines

An interrupt is any service request that causes the CPU to stop its currently executing process and execute instructions to service the request. The driver interrupt routine is responsible for determining the reason for the interrupt, servicing the interrupt, and waking up any base-level driver processes sleeping on the interrupt completion.

Following are a few of the routines commonly used for interrupt handling.

Table 1-4: Interrupt Handling Routines

Call	Description
`biodone`	Release buffer after block I/O and wake up process
`intr`	Process a device interrupt
`spl`	Block/allow interrupts on a processor

For more information about these and other calls and functions, refer to the DDI/DKI portion of the *Device Driver Reference*.

Input/Output Control Routines

I/O control commands can be used to do many things, including

- implement terminal settings passed from **getty**(1M) and **stty**(1)
- format disk devices
- implement a trace driver from debugging
- clean up character queues

To implement I/O control commands for a driver, two steps are required.

1. Define the I/O control commands and the associated value in the driver's header file
2. Code the driver `ioctl` routine to define the functionality for each I/O control command in the header file

Following are a few of the routines commonly used in I/O control.

Table 1-5: I/O Control Routines

Call	Description
biodone	Release buffer after block I/O
biowait	Suspend I/O process, pending completion of block I/O
physiock	Validate and issue raw I/O request
repinsb	Read bytes from I/O port to buffer
repoutsb	Write bytes from I/O port to buffer

For more information about these and other calls and functions, refer to the DDI/DKI portion of the *Device Driver Reference*.

Error Handling Routines

Error handling is one of the most important functions required in a device driver. Drivers must handle any error condition, or the consequences might be severe. For example, a stray interrupt should be a trivial event, but could panic the system if the driver is not prepared to handle it. A system panic can cause data corruption and physically damage the system.

When an error occurs, the driver can do one or more of the following:

- Write the error condition to a structure so the driver knows about it. Usually, at base level, the error is recorded in the **b_error** member of the **buf**(D4DK) structure.
- Retry the process. The error might be a transient problem. Some hardware device boards have retry capabilities; let these boards perform the retry. However, if the error is software related, the driver must decide how many times to retry.
- Report the error to a system error log. If the error is severe, take the faulty hardware out of service to minimize the damage and keep the system running normally.
- Report the error to the system administrator, either by printing it on the system console or by writing it to **putbuf** (to be reviewed with the **crash**(1M) utility).
- Send a signal to the user process.
- Panic the operating system.

In the following table are a few of the common error handling routines.

Table 1-6: Error Handling Routines

Call	Description
`cmn_error`	Write error message to console, `prtbuf`, or both
`geterror`	Retrieve error number from buffer header
`psignal`	Send signal to a single process

For more information about these and other calls and functions, refer to the DDI/DKI portion of the *Device Driver Reference*.

Driver Environment

Installation and Configuration

For a driver to be recognized as part of the UNIX system, information about what type of driver it is, where its object code resides, what its interrupt priority level will be, and so on must added to the system and stored in the appropriate system configuration files.

A device driver is added to a working UNIX system in four basic steps.

Configuration Preparation — Requires the driver writer to prepare a Driver Software Package (DSP), which includes the `Driver.o` object module (providing the actual driver code), package installation and removal scripts (both written by the driver writer), and various components (such as the driver's `Master` and `System` file definitions) used for system shutdown, configuration, and initialization.

Installation — Installs the driver's DSP, updates the system configuration files, and prepares for generating a new kernel.

Configuration — Invoked by shutting down and rebooting the system. During the reboot, the system uses information from the modified system configuration files to create special files in `/dev`, and the entries for the new driver in the system initialization tables, switch tables, and interrupt vector tables.

Loadable drivers can be configured into the kernel while the system is running, without rebooting the system and rebuilding the kernel. See Chapter 2, "Loadable Drivers", for a description of the loadable driver installation and configuration procedures.

Initialization — The driver itself is then initialized as part of the kernel when the system is reinitialized.

Drivers are installed and configured using a set of utilities called the Installable Driver Tools (idtools). Chapter 3, "Driver Installation and Tuning", gives details about how to install and configure drivers, and how the system is initialized.

Master and System Files

Two files are the source of some of the more important configuration information needed to make a driver part of a running system: the **Master** and **System** files.

Master File

A driver's **Master** file describes all of the devices supported by the driver than can possibly be configured into the system. Once the driver is installed, its **Master** file resides in the directory **/etc/conf/mdevice.d**. This directory contains a separate **Master** file for each installed device driver.

Configuration data defined in the **Master** file includes the names of the driver's entry point routines, and an alphanumeric prefix (assigned by the driver writer) that is prepended to the names of the driver's routines in the system tables. The prefix enables the kernel to distinguish one driver's routine names (and other variables) from another's, thereby avoiding conflict with other variables in the system with the same name. For example, a RAM disk driver may have been given a prefix of **ram_**, which would result in routines named **ram_open**, **ram_init** and so on. For more information, see the **prefix**(D1DK) manual page.

The **Master** file may also contain the driver's major number, and various flags that define the specific characteristics of that driver (for example, whether it is a character or block driver). During installation, the idtools will assign a major number if the driver's **Master** file doesn't specify one. For more information, see the **Master**(4) manual page.

System File

A driver's **System** file provides information needed to configure one or more devices supported by the driver into the next kernel to be built. Once the driver is installed, its **System** file resides in the directory **/etc/conf/sdevice.d**. This directory contains a separate **System** file for each installed device driver.

Configuration data defined in the **System** file includes a flag that indicates whether or not the driver ought to be incorporated into the kernel, and various values used by the kernel to interact with the driver's interrupt handler. For more information, see the **System**(4) manual page.

Driver Header Files

Driver source code must contain some standard "**include**" files that allow the driver access to system utilities and data structures used to return information to the kernel.

The description of each kernel utility function in the DDI/DKI manual pages indicates which header files must be included in a driver that uses that function. The list below identifies a few of the more commonly used **include** files

<sys/types.h>	Defines basic system data types.
<sys/param.h>	Defines fundamental system parameters.
<sys/signal.h>	Defines system signals. If the driver sends signals to user processes, it must include this file.
<sys/conf.h>	Defines device switch tables. This file is needed for the driver to define its **devflag** value.
<sys/file.h>	Defines file structures. This file is needed if the driver uses control flags such as "no delay" (**FNDELAY**).
</sys/buf.h>	Defines the **buf** (system buffer) structure. This file is needed if the driver uses the system buffer pool.
<sys/kmem.h>	Defines the Kernel Memory Allocator. This file is needed if the driver allocates memory for buffers out of the common memory pool.
<sys/ddi.h>	Defines Device Driver Interface (DDI) routines. This header file is required and must come last in the list of included header files (see exception below).
<sys/ddi.i386at.h>	Defines functions and symbols specific to the UNIX system for the Intel 386 architecture. If this platform-specific include file is used, it must come last in the list of included header files, after the **ddi.h** header file.

Sample Block Driver

The example driver described in this section is similar, in most of its parts, to all block drivers. It is a RAM disk driver (a software driver), which uses an area of memory for mass storage, but has no hardware to control. Consequently, it doesn't have to recognize or respond to interrupts (a major complication).

The RAM driver example illustrates the general structure of real disk drivers at only one level, called the base level. The base level includes the routines responsible for servicing the I/O request from the user process. The other level, called the interrupt level, responds only to requests for servicing hardware (non-existent for a RAM disk).

The work of the base level of a RAM disk driver is to open a file system, provide access to it, and close it when necessary. The entry-point routines required for these activities are **open**(D2DK), **strategy**(D2DK) and **close**(D2DK). The only other part of the RAM disk driver is the initialization routine **init**(D2D).

Each routine is illustrated (with pseudo-code) in the pages that follow. After the pseudo-code is a brief discussion of every line of the pseudo-program. The numbers in parentheses (before the lines of pseudo-code) are referenced by the section headers below, to indicate which line is being explained in that section. In the four sections that follow, code fragments from a working driver are included to help illustrate the concepts being described.

Initialization

Not all drivers have **init**(D2D) routines; some have nothing to initialize and others defer initialization to the **open**(D2DK) routine. In most cases, it doesn't matter if variables are zeroed in an **init** or an **open** routine. On the other hand, the system should be informed at the time of initialization if, for example, a disk drive is off-line.

Software drivers typically have little to initialize because no hardware is involved. In fact, some software drivers have completely empty **init** routines. Memory may be allocated as a simple two-dimensional array in the **open** routine. But even if no **init** routine is needed, the driver must have an entry point routine in the switch table. In the following pseudo-code for a software driver (Figure 1-4), required initialization processing is minimal. Some memory must be allocated and initialized, and a warning must be issued if the allocation fails.

Figure 1-4: Pseudo-code for init Routine

```
(1)        include header files
       init(dev)
(2)          if (memory can be allocated)
                allocate memory
                initialize memory
(3)             print informational message
             else
                 print warning message
```

Driver Header Files (1)

The first file in the list of header files included in driver code should be **sys/types.h** because many of the other header files use the type definitions it contains. In the **init** routine, the device number passed in as an argument is declared to have the type **dev_t**, which is an alias for a unsigned long integer. Simple data types are abstracted to these types to enhance driver portability.

Most drivers will need to include a minimum of 5 to 10 header files and some may have more than 20.

Memory Allocation (2)

The function used to allocate memory is **kmem_alloc**(D3DK). **kmem_alloc** accepts as an argument the number of bytes to be allocated and a flag that indicates whether the caller is willing to sleep waiting for the memory to be allocated. The **kmem_alloc** manual page also tells you what conditions must exist for the allocation to succeed, how different types of failures are handled, and which header files must be used.

Messages (3)

Another useful library function is **cmn_err**(D3DK). The **printf**(3S) library function cannot be used in driver code; instead, the function **cmn_err** is used for all types of messages, from the merely informational to those reporting severe errors. The first argument to this function is a constant used to indicate the severity level, the second is the formatted message string, and the third is an optional set of arguments passed with the message being displayed. For example, the following statement could be used to report why the initialization failed.

```
cmn_err(CE_WARN,"prefixinit: kmem_alloc cannot allocate %d buffers", BUFS);
```

The **cmn_err** function can also be used to shutdown or panic the system when serious errors are detected. For example, if a hardware driver is unable to allocate private buffer space there is probably sufficient reason to halt system initialization. When this condition is detected, the next statement should be

```
cmn_err(CE_PANIC,"prefixinit: Buffer space unavailable");
```

Other init Responsibilities

A working driver for a hardware device (for example, a disk drive) does not have an **init** routine as simple as the one shown earlier. The additional processing required may include some of the following

- Check to see if the devices under the control of the driver are actually on-line.
- Check for the correct number of subdevices.
- Set each device's interrupt vector to correspond to the system's interrupt vector table.
- Set the virtual-to-physical address translation.
- Set device-specific parameters to default values. These parameters include values for the number of tracks, cylinders and sectors.

Base-Level Operation

The base-level, entry-point routines do most of the work of the driver. These are the routines that respond to user I/O requests, expressed as system calls. The kernel then interprets the system call, and, in turn, calls one of the driver's entry-point routines.

There is not a one-to-one correspondence between system calls and driver routines. For example, on a multiuser system more than one user process may have opened a device. The kernel calls the driver **close** routine only when the last of these user processes issues the **close** system call. A user's read or write request results in a call to the block driver's **strategy** routine.

The open Routine

When a user process issues an **open**(2) system call, the file to be opened is most often a regular file, which is generally opened to read or write text or data. However, the driver **open**(D2DK) routine is opening the device, which looks like a file on a UNIX system.

- The special device file identifies which switch table (block or character) to look in for the driver open routine.
- After the correct switch table is identified, the major number is used to find the corresponding **open** routine.

Finally, when the open routine is called (Figure 1-5), it is passed a pointer to the device number and the flags indicating the type of open (read only, create new file, and so on).

Figure 1-5: Pseudo-code for open Routine

```
open(device number, flags, type, credentials)

     if (minor device number is invalid)
        return ENXIO
     else
        set up buffer to read the superblock
        call strategy
```

Each of the following sections covers the issues involved in implementing the processing represented by a line of pseudo-code. Most sections will also give an actual code sample (in the C language) to illustrate typical driver coding style.

Validating the Minor Device Number

The device number contains both the major number (identifying the driver) and the minor number (identifying the sub-device). The major number has already been used as an index into the switch table to call the driver, so there is usually no need to check its value. If the driver is using the multiple major number feature (described later in the guide), it should verify that the number is within the range the driver expects to use. The major number can be extracted from the device number via the **getmajor**(D3DK) function.

The minor number can be extracted via the **getminor**(D3DK) function. In the example below, the minor number is checked against a driver-defined constant **MAXDEV** If the minor number is not a simple unit number, but also contains components (a high-order bit, for example) that define the type of access to the physical device, the driver **open** routine should extract each component and verify that the unit number is valid and the type of access is appropriate. Sometimes physical access to the device will be needed to verify its actual presence on the bus. The **open** routine should return an error number if the **open** should not be permitted.

```
if (getminor(dev)) > MAXDEV)
        return (ENXIO);          /* No such device */
```

Returning Errors to the Calling User Process

When a driver needs to report an error to the user process that caused the call to a driver entry point, the usual method is to return an error number as the return value of the driver entry point. If there is no error to return, the value 0 should be returned. In the example above, when the minor number passed to the driver open routine was out of range (greater than the driver-defined **MAXDEV**), the value **ENXIO** is returned.

Driver error numbers are defined in the header file **sys/errno.h**, and are described in **errnos**(D5DK). The general algorithm for the driver entry points that correspond to system calls (namely **read**, **write**, **open**, **close**, **ioctl**, **poll**) is

```
/* verify arguments and perform entry point processing */

if (error condition or invalid arguments) {
        /* see entry point-specific documentation for appropriate
         * error numbers
         */
        return (error number);
}

return (0)
```

Setting Up a Buffer

The kernel buffer cache is a linked list of buffers used to minimize the number of times a block-type device must be accessed. A block driver does not read or write directly to the disk, but rather to the buffer cache.

The section "The strategy Routine" below explains how the driver reads and writes blocks. This section introduces the buffer header, that part of the buffer structure used to identify where the data came from. The structure is called **buf**(D4DK) and is defined in the file **buf.h**.

The Buffer Header

The buffer header is the structure used by the kernel's file system and virtual memory subsystems to pass I/O requests to disk drivers. These subsystems set the following fields in the **buf** structure to tell the driver what to do

b_dev The device number. This is a composite value, made up of both the major and minor number. It is used to identify the RAM device.

b_bcount The number of bytes to be transferred. This number can vary in value, so the driver writer must never assume a standard value for this field.

b_blkno The block number to access on the device specified by **b_dev**. Note that this is a logical number; if a physical disk is split into two logical sub-devices (each logical part has its own minor number), the block number refers to the block within the particular logical sub-device, not the physical block on the disk.

The block size is given by the constant **NBPSCTR** in **sys/param.h** and has nothing to do with the block size being used for a particular file system (s51K versus s52K, for example).

b_error Should be initialized to 0 by the driver and set to an error number if there is a failure or if the request is invalid. The flag **B_ERROR** in **b_flags** (see below) should also be turned on.

b_flags Values are OR'ed into this member (allowing more than one value to be on at a time). In particular, the driver will want to check whether the flag **B_READ** is set. If not set, then the I/O request is a write operation.

For more information, see the **buf**(D4DK) manual page.

Other open Routine Responsibilities

Like the **init** routine, the **open** routine for a RAM disk driver is simpler than for a hardware device. Other functions a hardware **open** routine may perform are

- initialize error logging
- initialize the disk defect table
- read the volume table of contents (VTOC) and the defect table
- read the physical description sector

For more information, see the **open**(D2DK) manual page.

The strategy Routine

The **strategy**(D2DK) routine may be called from the **open** routine to read state information (such as the VTOC) from the disk (Figure 1-6). More often, **strategy** is called in response to a system I/O request. This is the main work of a block device driver, and **strategy** is the routine that does it. To transfer data, the **strategy** routine is passed a pointer to a buffer header.

Figure 1-6: Pseudo-code for strategy Routine

```
include header files

strategy(bp)

   (1)   if (block number is out of range)
            write error to buf structure and set B_ERROR
            return

   (2)   if (I/O request is for read)
            read block of data
         else
            write block of data

   (3)   call biodone
         return
```

For more information, see the **strategy**(D2DK) manual page.

Check for Valid Block (1)

As part of the kernel, the RAM disk driver has access to any part of memory, and so it is very important to make sure that reading and writing of data is confined to the area allocated for the RAM disk. The most basic checking uses the **b_blkno** member of the buffer structure to make sure the requested block is within range. (**RAMBLKS** is the number of blocks in the RAM disk. Because the first block number is **0**, the block number equal to **RAMBLKS** can be calculated as the number of blocks in the RAM disk plus one.)

```
if (bp->b_blkno < 0 || bp->b_blkno+1 > RAMBLKS)
  bioerror(bp, ENXIO)
```

If the I/O request is for a block beyond the end of the disk, the driver must further check to see if a read or a write is requested. For a read, the number of unread bytes is reported by assigning the value of **b_bcount** to **b_resid,** which is passed by the system as a return value to the **read** system call.

```
if (bp->b_blkno > RAMBLKS && bp->b_flags & B_READ)
  bp->b_resid = bp->b_count;
```

The read status is tested by logically **AND**ing the **b_flags** member with the value **B_READ.** If the test fails, the I/O request is assumed to be a write. Any attempt to write beyond the end of the RAM disk must be denied, and an error reported.

```
bp->b_error = ENXIO;
bp->b_flags |= B_ERROR;
```

Reading and Writing Data (2)

Several different functions are available for moving data. Transfer can be between user space and the driver (with **copyin** and **copyout**). But the RAM disk and the driver **buf** header are both in kernel space, so the **bcopy**(D3DK) function is used. The three arguments to the function are the source of the data, the destination, and the number of bytes transferred.

```
if (bp->bflags & B_READ)
        bcopy(disk_addr, b_un.b_addr, bp->b_bcount);
else
        bcopy(b_un.b_addr, disk_addr, bp->b_bcount);
```

The biodone Function (3)

When the data transfer is complete, the **strategy** routine calls the **biodone**(D3DK) function. Hardware drivers use **biodone** to awaken sleeping processes. (This is not required for pseudo-devices.) The RAM driver uses this function to release the buffer block and to set the **b_flags** member to **B_DONE**. The **biodone** function is called with a single argument, the pointer to the buffer header.

```
biodone(bp);
```

For more information, see the **biodone**(D3DK) manual page.

The close Routine

Many drivers (even some hardware drivers) will have empty **close**(D2DK) routines. Even though it does nothing, the address of the empty routine is entered into the switch table.

```
close( )
        {
        }
```

If not empty, a **close** routine may be responsible for unlocking the device (if locked by the **open**(D2DK) routine), flushing buffers, making sure the device does not contain a mounted file system, and reinitializing its data structures.

Because more than one process may have opened the device, the **close** routine is not called if any process still has the device open. The way in which a file was opened may affect how it should be closed, so one of the arguments to the **close** routine is taken from the **file** structure (declared in **file.h**).

For more information, see the **close**(D2DK) manual page.

Driver Development

The rest of this chapter reviews a variety of steps and guidelines programmers should keep in mind when planning and developing device drivers.

Basic Steps for Creating a Driver

Device driver development requires more up-front planning than most application programming projects. At the very least, testing and debugging are more involved, and more knowledge about hardware is required. The following steps can be used as a general guide to driver development.

Preparation

- Learn about the hardware. Most of the information you need can be found in the documentation for the device, and should include
  - how the device sends interrupts
  - the range of addresses of the hardware board
  - return codes and software protocols recognized by the device
  - how the device reports hardware failures
- Test the hardware to make sure it is functioning. This is especially important for a newly-developed device.
- Design the software. Even though the overall structure of a driver is not the same as an application program, good structured design remains important. Data flow diagrams, functional specifications, and structure charts are all useful tools in driver development. Design documents should cover not only the driver contents, but also the contents of any utility programs that will be used with the driver.
- Select a software maintenance and tracking utility, such as the source code control system (SCCS) described in the **sccs**(1) manual page.

Implementation

- Write and install a minimal driver. It is very helpful to test driver code from the earliest stages, and to verify that it can be installed. A minimal driver might be one that simply uses the `cmn_err` function to send a "`hello, world`" message to the system console. See Chapter 3 for a detailed guide to driver installation.
- Write base-level routines before interrupt-level routines.
- If applicable to the device, write and test any associated firmware.
- Develop utilities such as disk formatting, network administration, and diagnostic programs at the same time as the driver.

Follow-up

- As much as possible, use the testing phase to create error conditions that exercise the driver's ability to handle them.
- Evaluate the driver's performance both in isolation and in a production environment where other drivers are installed. Regression testing should be performed to ensure that a new device driver does not affect other system functionality.
- Make sure documents affected by the creation of the driver are updated. These may include operator and diagnostic manuals and sales or ordering information. If the driver is to be installed by a customer, write and test installation and de-installation packages, as described in Chapter 3.

Commenting Driver Code

Good practice in commenting driver code is the same as for any type of programming. Because driver code can be extremely difficult to maintain without adequate comments, these guidelines are included here.

- Each file should have a comment block at the beginning, describing the type of file functions and the services they perform. List the functions that call them and the functions they call. For a hardware driver, describe the hardware, including version numbers and hardware strapping values.
- Describe each global data structure or type declared, including its possible range of values. Describe the protocol, if any, used to access it (such as flag-setting). If it is useful, describe the functions that access structures, including those that are in other files.

- Each routine should have a comment block at the beginning describing what it does, how it does it (what are the algorithms or strategy), assumptions about the environment when it is called (processor interrupt priority level, outstanding I/O jobs, and so forth), and what global variables are used.
- Each line that declares an argument to the routine should have a comment.
- Every local variable should be explained.
- Each loop or `if` test should have a comment to explain the exit condition.

Layered Structure

Hardware drivers will be easier to port and maintain if structured in layers.

- Separate the higher-level protocol functionality from the low-level, machine-dependent routines. The high-level sections can be readily ported, leaving only the low-level sections to be rewritten. If machine-specific code is not isolated, all code may need to be rewritten to run on another processor.
- When your driver accesses system structures such as the system buffer structures, use the standard functions included in the DDI/DKI. Using non-standard functions with standard structures can degrade the performance of other drivers on the system and will impact portability and forward compatibility.

Driver Functions

A device driver is made up of entry-point routines that call standard DDI/DKI functions and subordinate routines written for the driver. Here are some things to consider when using these functions and routines

- Standard functions, especially for timing and data allocation, are less likely to degrade system stability and performance than similar routines coded in the driver.
- When subordinate routines must be written, declare them `static` to prevent name conflicts with other drivers. In general, define as few global names (both functions and names) as possible. To make the driver easier to maintain, use the driver prefix when naming subordinate routines, even though the `static` declaration makes this step unnecessary.

Utilize Board Intelligence

Many peripheral devices are intelligent, meaning that they contain their own microprocessor that can hold driver code. For optimal performance and portability, take full advantage of the board's intelligence by writing a firmware driver that provides the basic functionality of the board, then accesses the firmware driver from within the UNIX system driver.

With intelligent devices, some of the control for a device or controller may be in code running on the controller board rather than in the driver running in kernel memory. The code for the controller board may be in firmware or may be downloaded to controller RAM, for example, at system boot time.

If the device never needs to work in a non-UNIX system (firmware) mode, it is not necessary to use firmware for anything more than diagnostics. You may also want to include in firmware a basic subset of the protocol necessary to talk to the host processor directly, such as the memory management protocol. Proper use of firmware can enhance the features, portability, and performance of your device.

2 Loadable Modules

Introduction

UNIX System V Release 4.2 supports Dynamically Loadable Modules (DLM). This feature allows you to add a device driver to a running system without rebooting the system or rebuilding the kernel.

The DLM feature

- reduces time spent on driver development by streamlining the driver installation process
- makes it easier for users to install drivers from other vendors
- improves system availability by allowing drivers to be configured into the kernel while the system is running
- conserves system resources by unloading infrequently used drivers when they are not needed (when needed in the system, DLM loads the drivers from disk)
- gives users the ability to load and unload drivers on demand
- gives the kernel the ability to load and unload drivers automatically
- requires drivers that are going to be configured into the system as loadable modules to be converted to loadable form

The discussion of DLM that follows contains two parts.

The first part provides an overview of the DLM feature from the driver writer's perspective. Among other things, this part explains how DLM creates a kernel that is different from the statically configured kernel you may be accustomed to working with. It also describes the different ways loadable modules can be loaded and unloaded, and provides an overview of how the DLM loading and unloading mechanism works. This background information should prove useful to you when you have to perform tasks such as debugging your loadable driver.

The second part explains how to convert your non-loadable driver to be a loadable driver. This part presents information you will need to write the initialization code that lets DLM dynamically connect your loadable driver to the rest of the kernel. It also tells you how to install your driver as a loadable driver, how to configure your loadable driver into a running system, and to how load it. Information about debugging a loadable driver is also provided.

The DLM Mechanism

Loadable Module Types

Since this book is about device drivers, this chapter focuses on loadable device drivers. However, you should be aware that the DLM feature supports loading and unloading of a variety of kernel module types.

Types of modules that can be loaded include

- device drivers (block, character, STREAMS and pseudo)
- Host Bus Adapter (HBA) drivers
- Direct Coupled Device (DCD) controller drivers
- STREAMS modules
- file systems
- miscellaneous modules—for example, modules containing code for support routines shared among multiple loadable modules which are not needed in the statically configured kernel

Although the discussion focuses on device drivers, the information being presented in this chapter applies—in a general way—to all loadable module types.

The Difference between Static Modules and Loadable Modules

In previous releases, all kernel modules were maintained in individual object files (`.o` files) so they could be conditionally included or excluded from the kernel, depending on whether or not the features they supported were required in the system. The conditional nature of this arrangement meant that when you wanted to add a new module or remove an existing module, you had to relink the entire kernel and reboot the system to cause your new kernel configuration to take effect.

With DLM, some modules continue to be linked to the kernel in the traditional manner. Kernel modules that are configured this way are called static modules. A static module is, by definition, non-loadable. That is, the module remains linked into the kernel at all times because either it is always required in the system (like the boot hard disk driver), or it is used so frequently or consumes so few resources (like the user terminal pseudo-device driver) that it makes sense to keep the module continuously configured.

Other modules—modules that are not required, are used infrequently, or consume large amounts of resources—can be configured so they can be included or excluded from the kernel dynamically, without a system shutdown and reboot. These modules are called loadable modules.

Loadable modules are also maintained as individual object files, but they are not statically linked to the kernel. Instead, they are linked into the kernel when they are needed and unlinked when they are no longer in-use. Floppy disk drivers and mouse drivers are two examples of kernel modules that are typically configured as loadable modules.

Overview of the Load Process

When a loadable module needs to be added to the system, the DLM mechanism reads the module's object file on disk and copies the module into dynamically allocated kernel memory.

Once the module is in memory, DLM relocates the module's symbols and resolves any references the module makes to external symbols. DLM then executes special code in the module (called "wrapper" code) that enables the module to initialize itself dynamically.

When module initialization is complete, DLM executes code specific to the loadable module type. This code logically connects the module to the rest of the kernel.

Overview of the Unload Process

The unload process undoes what was done during the load process.

First, the DLM mechanism executes code specific to the loadable module type that logically disconnects the module from the rest of the kernel. Once in the module is disconnected, DLM then executes the module-supplied wrapper code that enables the module to clean up for termination. When clean-up is complete, DLM releases the memory allocated for the module.

The Difference between a Demand Load and an Auto Load

Two types of events can cause a module to be loaded or unloaded by the DLM mechanism: a demand-load/unload request or an auto-load/unload event.

Demand Load

A demand load is a user request, made using the **modadmin**(1M) command, to add a loadable module to the running system.

If the module depends on other loadable modules and these modules are not currently loaded, DLM will automatically load these modules during the load process.

NOTE

For the initial release of the DLM feature, loadable DCD controller drivers cannot be demand loaded. They must be auto loaded by the kernel as they are needed.

Note also that if the DCD module is a loadable module, all configured DCD controllers must be also be loadable. Conversely, if the DCD module is a static module, all configured DCD controllers must be also be static.

Demand Unload

A demand unload is a user request, made using the **modadmin**(1M) command, to remove a loadable module from the running system.

If the module is not being used when the request is made, and if no other loaded module depends on the module, DLM will unload it. If the module is being used, or if another loaded module references symbols defined in the module, DLM does not unload the module. Instead, DLM adds the module to a list of modules that are candidates for the next auto unload.

Auto Load

An auto load occurs when the kernel determines that the functionality provided by a particular module is required to perform some task. For example, the kernel would call DLM to auto load a loadable device driver on the first **open** of any of the driver's configured devices. A loadable STREAMS module would be auto loaded on the first **I_PUSH** of the module. During an auto load, DLM also loads any modules that the module being loaded depends upon, as it does during a demand load.

NOTE

For the initial release of the DLM feature, loadable HBA drivers cannot be auto loaded. HBA drivers can, however, be demand loaded using the `modadmin` command, or demand loaded by `init`(1M) via the `idmodload`(1M) command during a system reboot.

Note also that, once loaded, an HBA driver remains loaded until the next system reboot (no DLM unload mechanism exists for HBA drivers).

Auto Unload

An auto unload can occur when the kernel determines that the amount of memory available to it is low. To deal with this shortage, the kernel calls DLM to attempt to unload any modules that have become candidates for unloading. Modules become candidates for auto unloading when they are inactive, they have not been accessed for some predetermined amount of time, and no other loadable modules depend on them.

For example, a loadable device driver would become a candidate for auto unloading on the last `close` of all its configured devices, and a loadable STREAMS module would become a candidate for auto unloading on its last `I_POP`. The amount of time that must elapse before inactive modules are considered candidates for auto unloading is controlled by the value of the global tunable parameter `DEF_UNLOAD_DELAY` in `/etc/conf/mtune.d/kernel`. Individual modules can override the value of the global auto-unload delay by specifying their own auto-unload delay value in their `Mtune`(4) files.

NOTE

On a demand unload request, the auto-unload delay parameter value is ignored.

If the attempt to auto unload a module is successful, the memory allocated for the module is reclaimed. Unloading continues until the amount of available memory reaches a predetermined high-water mark or the list of unloadable candidates is exhausted.

NOTE

Modules that are demand loaded cannot be auto unloaded. If a demand-loaded module is no longer needed in the system, it must be demand unloaded.

Making Modules Loadable

The following sections explain how to convert your non-loadable driver to be a loadable driver.

Coding a Wrapper

The first step in converting a non-loadable driver to a loadable driver is writing some special initialization code called a "wrapper."

Each loadable module is required to supply the DLM mechanism with a wrapper. The wrapper "wraps" a module's initialization and termination routines with special code that enables DLM to logically connect and disconnect the module to and from the kernel "on the fly" while the system is running.

The wrapper consists of function definitions and initialized data structures.

Wrapper Functions

For a device driver, the wrapper functions can include

prefix`_load` The `_load` entry point is called by the DLM mechanism once the driver has been loaded into memory and link edited into the kernel. The `_load` routine handles any initialization tasks the driver must perform prior to being logically connected to the kernel. Typical initialization tasks performed from `_load` include acquiring private memory for the driver, initializing devices and data structures, and installing device interrupts. This entry point is optional, and is described on the `_load`(D2D) manual page.

prefix`_unload` The `_unload` entry point is called by the DLM mechanism once the driver has been logically disconnected from the kernel. The `_unload` routine handles any clean-up tasks the driver must perform prior to being removed from the system. Typical clean-up performed from `_unload` include releasing private memory acquired by the driver and removing device interrupts. This entry point is optional, and is described on the `_unload`(D2D) manual page.

prefix`init` The `init` entry point is called by the driver's `_load` routine to perform any setup and initialization the driver must do before interrupts are enabled. If you are converting a non-loadable driver to make it loadable, you will probably find that you can use your static driver's `init` routine in the loadable version of the driver. This entry point is optional, and is described on the `init`(D2D) manual page.

prefix`start` The `start` entry point is called by the driver's `_load` routine to perform any setup and initialization the driver must do after interrupts are enabled. If you are converting a non-loadable driver to make it loadable, you will probably find that you can use your static driver's `start` routine in the loadable version of the driver. This entry point is optional, and is described on the `start`(D2DK) manual page.

prefix`halt` The `halt` entry point is called by the DLM mechanism. If the driver is loaded at the time the system is shut down, DLM will call the driver's `halt` routine to shut down the driver when the `halt` routines for the statically configured kernel modules are called. If you are converting a static driver to make it loadable, you will probably find that you can use your static driver's `halt` routine in the loadable version of the driver. This entry point is optional, and is described on the `halt`(D2DK) manual page.

`mod_drvattach` The `mod_drvattach` routine is called by the driver's `_load` routine to add the driver's interrupts to the running system. Since interrupts are enabled upon return from `mod_drvattach`, you should make sure your driver's `_load` routine calls its `init` routine prior to calling `mod_drvattach`, and calls its `start` routine after calling `mod_drvattach`. This routine is optional, and is described on the `mod_drvattach`(D3DK) manual page.

`mod_drvdetach` The `mod_drvattach` routine is called by the driver's `_unload` routine to disable and remove the driver's interrupts from the running system. This routine is optional, and is described on the `mod_drvdetach`(D3DK) manual page.

Wrapper Data Structures

The wrapper data structures are initialized by the DLM mechanism using values taken from your driver's configuration files. These structures provide information needed during loading and unloading—such as the values needed to populate your driver's **bdevsw** or **cdevsw** switch table entries for the major device numbers it supports.

Note that your driver does not need to use any of the wrapper data structures directly, and your driver's wrapper needs only to point to these structures. However, if your driver requires interrupts to be added and removed, it will need to reference the **attach_info** structure in its **mod_drvattach** and **mod_drvdetach** wrapper routines. This structure contains the name of your driver's interrupt handler, and other related information. Although the **attach_info** structure is initialized by DLM using information taken from your driver's configuration files, as is done for the wrapper data structures, the **attach_info** structure is not linked to the wrapper data structures themselves.

Wrapper Macros

To aid you in generating a wrapper for your loadable driver (or other loadable module type), DLM provides a set of macros in **/sys/moddefs.h**. The macros are of the form

type(*prefix, load, unload, halt, description*);

The keyword *type* identifies the type of wrapper to be generated. Valid types are

MOD_DRV_WRAPPER	generates wrappers for device drivers, including block drivers, character drivers, STREAMS drivers and pseudo drivers
MOD_HDRV_WRAPPER	generates wrappers for Host Bus Adapter drivers, and any other driver type that does not require switch table entries, but does need to attach and detach interrupts
MOD_STR_WRAPPER	generates wrappers for STREAMS modules
MOD_FS_WRAPPER	generates wrappers for file systems
MOD_MISC_WRAPPER	generates wrappers for miscellaneous modules

The keyword *prefix* specifies the driver's prefix, as defined in the driver's **Master**(4) file, and described on the **prefix**(D1DK) manual page. The keywords *load*, *unload* and *halt* specify the names of the driver's **_load** routine, **_unload** routine, and (if the driver has one) its **halt** routine.

NOTE

For non-driver modules, the keyword *halt* is omitted from the wrapper macro coding.

The keyword *description* supplies a character string used to identify the driver.

Sample Wrapper Code

The following coding examples show some typical wrappers for the different loadable module types. Note that all loadable modules must include **`sys/moddefs.h`** in their wrapper definitions.

Figure 2-1 shows a sample wrapper for a device driver.

Figure 2-1: Device Driver Wrapper Coding Example

```
#include <sys/mod/moddefs.h>

extern    int       fdloaded; /* in fdbuf */

#define             DRVNAME             "fd - Floppy disk driver"

STATIC    int       fd_load(), fd_unload();

MOD_DRV_WRAPPER(fd, fd_load, fd_unload, NULL, DRVNAME);

STATIC    int
fd_load()
{
          fdinit();
          mod_drvattach(&fd_attach_info);
          fdstart();

          return(0);
}

STATIC    int
fd_unload()
{
          mod_drvdetach(&fd_attach_info);
          fdloaded = 0;

          return(0);
}
```

Figure 2-2 shows a sample wrapper for a Host Bus Adapter driver. Notice that, since HBA drivers cannot be unloaded, the **_unload** routine in the example simply returns the error number **EBUSY**.

Figure 2-2: Host Bus Adapter Driver Wrapper Coding Example

```
#include <sys/mod/moddefs.h>

#define             DRVNAME   "adsc - Adaptec SCSI HBA driver"

STATIC    int       adsc_load(), adsc_unload();
int       adscinit();
void      adscstart();

MOD_HDRV_WRAPPER(adsc, adsc_load, adsc_unload, NULL, DRVNAME);

static    int       mod_dynamic = 0;

STATIC int
adsc_load(c)
int c;
{
          mod_dynamic = 1;

          if( adscinit()) {
                    return( ENODEV );
          }
          mod_drvattach( &adsc_attach_info );
          adscstart();
          return(0);
}

STATIC int
adsc_unload()
{
          return(EBUSY);
}
```

Figure 2-3 shows a sample wrapper for a STREAMS module. Notice that the macro definitions for this non-driver module do not include a keyword for a **halt** routine name.

Figure 2-3: STREAMS Module Wrapper Coding Example

```
#include <sys/mod/moddefs.h>

int isoc_load(), isoc_unload();

int isocdevflag = D_OLD;

MOD_STR_WRAPPER(isoc, isoc_load, isoc_unload, "isoc - ISC socket emulation");

int
isoc_load(void)
{
        /* Module specific load processing... */
        int
        isoc_load(void)
        {
                /* Module specific load processing... */
                cmn_err(CE_NOTE, "!MOD: in isoc_load()");
                return(0);
        }


        int
        isoc_unload(void)
        {
                /* Module specific unload processing... */
                cmn_err(CE_NOTE, "!MOD: in isoc_unload()");
                return(0);
        }
```

Figure 2-4 shows a sample wrapper for a file system module. Notice that this file system module doesn't need to do any clean-up when it is unloaded, so its wrapper defines a NULL **_unload** routine.

Figure 2-4: File System Module Wrapper Coding Example

```
#include <sys/mod/moddefs.h>

STATIC int s5_load(void);

MOD_FS_WRAPPER(s5, s5_load, NULL, "Loadable s5 FS Type");

STATIC int
s5_load(void)
{

        inoinit();

        bzero((caddr_t)&s5fshead, sizeof(s5fshead));
        s5fshead.f_freelist = &s5ifreelist;
        s5fshead.f_inode_cleanup = s5_cleanup;
        s5fshead.f_maxpages = 1;
        s5fshead.f_isize = sizeof (struct inode);
        s5fshead.f_max = ninode;

        fs_ipoolinit(&s5fshead);
        return 0;
}
```

Figure 2-5 shows a sample wrapper for a miscellaneous module. Notice that, once loaded, this module wants to remain loaded, so its **_unload** routine always returns **EBUSY**.

Figure 2-5: Miscellaneous Module Wrapper Coding Example

```
#include <sys/mod/moddefs.h>

STATIC    int       clis_load(), clis_unload();

MOD_MISC_WRAPPER(clis, clis_load, clis_unload, "clist - character io");

static    int       mod_dynamic = 0;

STATIC int
clis_load()
{
          mod_dynamic = 1;
          cinit();
          return(0);
}

STATIC int
clis_unload()
{
          /*
           * This module can not be unloaded.
           */
          return(EBUSY);
}
```

Packaging a Loadable Module for Installation

Once you have written a wrapper for your loadable driver (or other loadable module type), you compile your driver in the normal way. Once you compile, you are ready to package the driver's object file for installation in its loadable form.

This section—and the sections on installation and configuration that follow—describe procedures that are specific to loadable modules. For information about the installation tools and procedures for both loadable modules and static modules, refer to the chapter "Installation and Tuning".

Master File Definitions for Loadable Modules

Loadable drivers can define two optional lines of configuration data in the **Master** component of their Driver Software Package (DSP):

$depend specifies the loadable modules on which the driver depends

$modtype defines a character string that identifies the driver type in error messages

If your loadable driver references symbols defined in other loadable modules, you must supply DLM with the names of these modules so it knows to load them when it loads your driver. You define the modules to DLM by listing them on the **$depend** line of your driver's **Master** file. You can specify all of the module names (separated by white space) on a single **$depend** line. You can also specify them individually, on multiple **$depend** lines.

The **$modtype** line in the **Master** file lets you define a character string that helps identify a driver that is loaded as a miscellaneous module type in DLM error messages. This string can be a maximum of 40 characters long, including all white spaces.

For a description of the **Master** file format, refer to the **Master**(4) manual page.

System File Definitions for Loadable Modules

To get themselves configured into a running system, all loadable drivers must identify themselves as loadable drivers in the **System** component of their DSP. Two types of **System** file entries are required for loadable drivers:

$loadable instructs the **idbuild**(1M) command to configure the driver into the system as a loadable driver

configure instructs **idbuild** to configure a specific device supported by this loadable driver into the system

If you want to configure your driver as a loadable driver, you must first define a **$loadable** line in the driver's **System** file that specifies the name of your driver. This line identifies your driver as a loadable driver type.

Next, you set a flag in the *configure* field (the second field) of the **System** file entry for each major device number supported by your driver. This flag indicates (**Y** or **N**) whether the device is to be configured into the system. Note that the *configure* field can be used to configure both loadable devices and static devices.

Note also that, in the future, if you want to statically link your loadable driver into the kernel, you will need to comment out the driver's **$loadable** line by inserting the character # in column one.

For a description of the **System** file format, refer to the **System**(4) manual page.

Mtune File Definitions for Loadable Modules

Loadable drivers can override the kernel's global auto-unload delay parameter values by supplying their own values in the **Mtune** component of their DSPs.

The global auto-unload delay values are defined in **/etc/conf/mtune.d/kernel** as

DEF_UNLOAD_DELAY *delay min max*

This says that, by default, any loadable module becomes a candidate for auto unloading when the module has not been accessed during the previous *delay* seconds. It also says that if your loadable driver wants to override the kernel's default auto-load *delay* value, you must specify a **DEF_UNLOAD_VALUE** that is greater than or equal to *min* seconds, and less than or equal to *max* seconds, in your driver's **Mtune** component.

The symbolic name of the driver's **Mtune** file delay variable must begin with the driver's PREFIX in full caps.

Checking the Configured Loadable Modules before Installation

DSP installation scripts often use the **idcheck**(1M) command to acquire information about the system into which a package is going to be installed. Most of the checks performed by the **idcheck** command return information about kernel modules that are installed. If you need information about the modules that are configured into the kernel when you are installing your loadable driver, you can use two **idcheck** options to inquire about the configured modules.

The **-y** option checks whether a named DSP is configured into the next kernel to be built by examining the *configure* field(s) of its **System** file entries. The **-p** option checks whether a named DSP was configured into the last kernel to be built by examining **/stand/unix**.

For more information, see the **idcheck**(1M) manual page.

Installing a Loadable Module

Loadable drivers are installed using the **idinstall**(1M) command with the **-a** option (add DSP components), the **-u** option (update DSP components), or the **-M** option (add or update out-of-date DSP components only). You use the **idinstall** command to install a loadable driver the same way you would use it to install a static driver; there are no special options or procedures for loadable modules.

For more information, see the **idinstall**(1M) manual page.

Removing a Loadable Module

Loadable drivers are removed using the **idinstall** command with the **-d** option, in the same manner as static drivers are removed.

However, once you have removed your loadable driver, you must remember to issue the **idbuild**(1M) command with no options. This way, **idbuild** can perform a deferred rebuild that adjusts any tunables affected by your removed module. The rebuild will occur later, on the next system reboot.

For more information about using **idbuild** with loadable modules, see the section "Configuring a Loadable Module" below.

Tuning a Loadable Module

If your loadable driver needs to modify any tunable parameter values, you must make these changes using the **idtune**(1M) command.

By default, parameters tuned using **idtune** do not take effect until the entire kernel is rebuilt and rebooted. When installing a DSP for a loadable driver that modifies the values of its existing tunable parameters in **/etc/mtune.d**, you probably want the driver's new parameter values to take effect immediately. To make the new values effective at the time you configure the driver into the running system (rather than at boot time), you can use the **idtune** command with the **-c** option.

Since the **-c** option modifies tunable parameters for loadable modules already configured into the kernel—in addition to modifying parameters for any loadable modules that will be subsequently configured—this option should be used with caution. You should take care to avoid introducing any inconsistencies between the tunables for the running kernel and those for your new loadable driver.

For more information, see the **idtune**(1M) manual page.

Configuring a Loadable Module

Once your loadable driver is installed, the next step is to configure it into the system using the **idbuild**(1M) command.

There are two ways you can configure your loadable driver using **idbuild**: a deferred build and an immediate build. If you don't want to configure your driver into the system that is currently running, you can invoke **idbuild** with no options, and your driver will be configured on the next reboot. If you do want to configure your loadable driver into the running system, you invoke **idbuild** with the **-M** option. This option configures your driver into the system immediately, without a reboot.

When no options are given, the **idbuild** command does not rebuild the kernel. It simply sets a rebuild flag and exits. The next time the system is rebooted, the reboot process checks this flag. Finding the flag set, the reboot process invokes **idbuild** with the **-B** option to rebuild the kernel and reconfigure all modules flagged as loadable in **/etc/conf/sdevice.d**. The new loadable modules are saved in **/etc/conf/modnew.d** instead of **/etc/conf/mod.d** and the new kernel is placed in **/etc/conf/cf.d/unix**. If the rebuild is successful, **idbuild** invokes the system shutdown and reboot process. During the reboot, the new kernel replaces the old kernel in **/stand/unix**, the directory **/etc/conf/mod.d** is removed, and **/etc/conf/modnew.d** is renamed to **/etc/conf/mod.d**.

With the **-M** option, **idbuild** configures your loadable driver into the running system immediately, so you don't have to wait for a reboot to be able to load it. Some of the tasks the **-M** option performs to configure your loadable driver include placing the driver's loadable image in the **/etc/conf/mod.d** directory, and creating the necessary nodes in the **/dev** directory. If your DSP contains an **Init** component, **idbuild** adds and activates your driver's **inittab** entries. When these tasks are complete, **idbuild** registers your driver with the kernel to make it available to the rest of the system.

For more information, see the **idbuild**(1M) manual page.

Loading and Unloading a Loadable Module

Loading the Module

Once your loadable driver is configured into the kernel, you are ready to load it using the **modadmin**(1M) command.

The **-l** option instructs **modadmin** to load a loadable module into the running system. For example, the command

```
modadmin -l lp
```

loads a line printer driver named **lp**.

If the **lp** driver references symbols in other loadable modules (as defined in the **$depend** line in its **Master** file), and some or all of these modules are not already loaded, **modadmin** loads them along with the **lp** driver. When loading completes, **modadmin** prints (on **stdout**) an integer *module-id* used to identify driver **lp**.

Querying the Module's Status

Once you have loaded your driver, you can view status information about the driver using the **-Q** or the **-q** option. For example, the command

```
modadmin -Q lp
```

requests status for the **lp** driver by specifying its module name, and the command

```
modadmin -q module-id
```

requests status for the **lp** driver by specifying the *module-id* returned by the **-l** option.

Information returned by the **-Q** and **-q** options includes the driver's auto-unload delay value, its reference count (the number of kernel modules currently accessing the driver), its dependent count (the number of loadable modules upon which the driver depends), and the pathname to its object file on disk.

Modifying the DLM Search Path

If you have placed your driver's object file somewhere other than in the default directory **/etc/conf/mod.d**, you need to give DLM the pathname to this location using the **modadmin** command with the **-d** option before you attempt to load your driver with the **-l** option.

For example, if you had installed the **lp** driver on a remote server in a directory named **/nfs/mod.d**, you would first use the command

```
modadmin -d /nfs/mod.d
```

to add the directory **/nfs/mod.d** to the search path DLM uses to locate loadable modules on disk.

Unloading the Module

The **-u** and **-U** options instruct **modadmin** to unload a module from the running system. For example, the command

```
modadmin -U lp
```

unloads the **lp** driver by specifying its module name, and the command

modadmin -u *module-id*

unloads the **lp** driver by specifying the *module-id* returned by the **-l** option.

If **lp** is currently in-use (that is, its reference count is not equal to 0), or if another loaded module references symbols in **lp** (that is, its dependent count is not equal to 0), the request to unload the **lp** driver will fail. If this occurs, DLM adds **lp** to a list of candidates for a subsequent kernel auto unload.

For a complete description of the **modadmin** command line options, refer to the **modadmin**(1M) manual page.

Debugging a Loadable Module

DLM Error Messages

DLM error messages are written to the kernel's **putbuf** message buffer. When a module fails to load, you can often determine the cause of the error by printing the messages in this buffer.

For example, if a demand load of a module fails with the error number **ERELOC** (indicating symbol referencing errors during relocation), the messages in **putbuf** give you the ability to identify the particular symbols that are causing the problem. This buffer can be examined while in the kernel debugger **kdb** by dumping its contents. For example, the command

```
putbuf 100 dump
```

dumps the first 256 bytes (100 hex) in the buffer. During normal system operation, DLM error messages can also be read from the **/dev/osm** node.

For information about **kdb**, refer to the **kdb**(1M) manual page.

Dynamic Symbols and kdb

As a consequence of the DLM feature, a dynamic symbol table is now maintained in kernel address space. The dynamic symbol table contains all global symbols defined in the static kernel—plus all global symbols defined in all currently loaded modules. The contents of the dynamic symbol table change as modules are loaded and unloaded; when a module is loaded, its symbolic information is added to the table, and when the module is unloaded, its symbolic information is deleted.

Note that the symbols defined in loadable modules are not known to **kdb** until they have been successfully relocated and resolved during loading. When debugging routines called during a DLM load operation (such as **_load**, **init** or **start**), it is useful to have access to the module's symbols as soon as possible.

The best way to do this in **kdb** is to break upon return from the DLM routine **mod_obj_load()** in **modld()**, and then single step until the symbol availability flag is set (about 10 instructions). Once available, the loadable module's symbols can be accessed in the same manner as you would access any other kernel symbol.

For information about the dynamic symbol table, refer to the **getksym**(2) manual page.

3 Driver Installation and Tuning

Introduction

For device driver writers, installation means different things. If you are installing a driver for a piece of hardware, for example, you'll have some hardware-related installation procedures to follow. When you install the driver you've written on your computer for the first time, you probably will be installing the driver without the installation scripts recommended for customer use. When you do create the device driver package for customers, called a Driver Software Package (DSP), installation will take on a different meaning.

This chapter discusses how to install device drivers using Installable Driver Tools (also known as idtools) and DSPs. Tuning and configuring, as it pertains to device drivers, is also covered, concentrating mainly on those details specific to device drivers, and on features new for this release of the UNIX system. This chapter also describes the idtools and tunable parameter commands that are used with device drivers.

For more information about software packaging, refer to the *UNIX Software Development Tools*.

Using idtools

Device drivers (and other types of kernel modules) are packaged, installed, and configured into the system using a collection of configuration files, commands, and scripts known as the Installable Driver Tools, or idtools. (They have also been known as the Installable Driver/Tunable Parameter (ID/TP) scheme and as the Installable Device Tools.)

It is important to note that the latest release of idtools has automated much of what used to be manual editing of driver configuration files. There are several benefits to automating this process, among them being decreased chances of total system failure because a single file has been lost or corrupted, fewer problems when installing a new driver, and a much simpler process for removing installed drivers.

Although you might create the configuration file without using idtools, once the file becomes part of a device driver, everything you do with the file from then on—from installing it, to rebuilding the UNIX system kernel, to removing the driver from the system—should all be done using idtools.

Detailed information on each of the idtools commands can be found in the Section 1M manual pages in the *Command Reference*.

idtools Enhancements for This Release

For UNIX System V Release 4.2, idtools has many enhancements that you need to know about, particularly if you have used idtools in previous releases, or if you are going to be installing or reworking DSPs created for previous releases.

Here's an overview of how the new idtools are going to affect you; we'll cover the details later.

Even if you are familiar with idtools, you should read the list below. idtools has changed in design and functionality in this release of the UNIX system.

- Once idtools installs your DSP components in **`/etc/conf`**, you should not edit the configuration files in this directory directly, the way you may have done so in the past. If you need to access files in **`/etc/conf`**, you should use the commands **`idinstall`**(1M), **`idcheck`**(1M), and **`idtune`**(1M).

- Each driver now has its own individual configuration files. For example, instead of using one large **mdevice** file containing definitions for every driver, there's now a directory, **/etc/conf/mdevice.d**, that contains a **Master** file for each driver, with file names that match the names of the individual drivers.
- The **Master**(4), **System**(4), and other file formats have changed. You should read the latest versions of the Section 4 manual pages on device driver configuration file formats.
- **Mtune** used to be a single file containing tunable parameters. Now, it has become the **mtune.d** directory, although the files in the directory have the same format that **mtune** used to.
- If you attempt to install a DSP containing a **Master** or **System**, component that is in the old-style format, idtools will automatically convert the file to the new-style format (except for the exec type) during the installation process. However, the conversion code will be removed eventually, so if you're creating a new DSP, you should use the new-style file formats.
- The **mfsys** and **sfsys** files (now obsolete) will be converted into **Master** and **System** files. However, this conversion code eventually will be removed, as well, so you should use the new format.
- The UNIX system kernel is now located in **/stand/unix**, instead of **/unix**. This change was made to get the kernel out of the root directory, and also to speed up the process of booting the system, because **/stand** is a bfs file system. This change was made with the UNIX System V Release 4.0.
- idtools supports installation and configuration of loadable modules without a system shutdown and reboot (see the chapter "Loadable Modules").

idtools Utilities and Commands

In the DSP, the **postinstall** script executes **idcheck**, **idinstall**, and **idbuild** to install the package and rebuild the UNIX system kernel. Manual pages for these commands are provided in the *Command Reference*. Details about the DSP component files (such as the **postinstall** script, **Driver.o**, **Master**, and so on) are covered later in this chapter.

idbuild

idbuild builds a UNIX system base kernel and/or configures loadable kernel modules using the current system configuration in *$ROOT*/*$MACH*/**etc/conf**.

Building a UNIX system kernel consists of three steps.

1. Configuration tables and symbols, and module lists are generated from the configuration data files.
2. Configuration-dependent files are compiled, and then are linked together with all of the configured kernel and device driver object modules.
3. If the loadable kernel module feature or a kernel debugger is enabled, kernel symbol table information is attached to the kernel.

The kernel is, by default, placed in *$ROOT*/*$MACH*/**etc/conf/cf.d/unix**.

If the kernel build is successful and *$ROOT* is null or "/," **idbuild** sets a flag to instruct the system shutdown/reboot sequence to replace the standard kernel in **/stand/unix** with the new kernel. Then, another flag will be set to cause the environment (device special files, **/etc/inittab** and so on) to be reconfigured accordingly.

If one or more loadable kernel modules are specified with the **-M** option, **idbuild** will configure only the specified loadable kernel modules and put them into the *$ROOT*/*$MACH*/**etc/conf/mod.d** directory. Otherwise a UNIX system base kernel is rebuilt with all the loadable modules reconfigured into the *$ROOT*/*$MACH*/**etc/conf/modnew.d** directory, which will be changed to **/etc/conf/mod.d** at the next system reboot if *$ROOT* is null or "/" [see **modadmin**(1M)].

If a loadable module has already been loaded, but to another major number range, you can either unload the module and then use **idbuild** with the **-M** option, or use **idbuild** without the **-M** option and reboot the system. (This assumes that *$ROOT* is null or "/.") If you attempt to use the **-M** option for a module already loaded at another major number range, **idbuild** will fail with error **ENXIO**.

When loadable kernel modules are configured with the **-M** option, **idbuild** also creates the necessary nodes in the **/dev** directory, adding and activating **/etc/inittab** entries if any **Init** file is associated with the modules, and registering the modules to the running kernel. This makes them available for dynamic loading without requiring a system reboot.

Base kernel rebuilds are usually needed after a statically linked kernel module is installed, when any static module is removed, or when system tunable parameters are modified. If you execute **idbuild** without any options and if the environment variable *$ROOT* is null or "/," a flag is set and the kernel rebuild is deferred to next system reboot.

idcheck

The **idcheck** command is used to obtain selected information about the system configuration. The **idcheck** command is designed to help driver writers determine whether a particular driver package is already installed or to test for interrupt vectors, device addresses, or DMA controllers already in use. The **idcheck** command is used in **postinstall** scripts that test for usable IVN, IOA, and CMA values and then instruct the user to set particular switches or straps on the controller board.

idinstall

The **idinstall** command is used by the DSP's **postinstall** and **preremove** scripts, and its function is to install, remove, or update a DSP.

idinstall is called by a DSP installation script or removal script to add (**–a**), delete (**–d**), update (**–u**), or get (**–g** or **–G**) device driver/kernel module configuration data. It can also be run from a kernel source makefile to make (**–M**) driver/module configuration data.

idinstall expects to find driver/module component files in the current directory. When components are installed or updated with **-a** or **-u** option, they are copied into subdirectories of the **/etc/conf** directory and then deleted from the current directory, unless the **-k** flag is used to keep them.

In the simplest case of installing a new DSP, the command syntax used by the DSP's Install script should be **/etc/conf/bin/idinstall** –**a** *module-name*. In this case the command requires and installs the DSP **Driver.o**, **Master**, and **System** components, and optionally installs the **Space.c**, **Stubs.c**, **Node**, **Init**, **Rc**, **Sd**, **Modstub.o**, **Sassign**, and **Mtune** components if those files are present in the current directory.

The **Driver.o**, **Modstub.o**, **Space.c**, and **Stubs.c** components are moved to a directory named **/etc/conf/pack.d/***module-name*. The remaining components are stored in directories under **/etc/conf**, which are organized by component type, in files named *module-name*. For example, the **Node** file would be moved to **/etc/conf/node.d/***module-name*, the **Master** file moved to **/etc/conf/mdevice.d/***module-name*, and the **System** file moved to **/etc/conf/sdevice.d/***module-name*.

idinstall -a requires that the module specified is not currently installed.

idinstall -u *module-name* performs an Update DSP (that is, one that replaces an existing device driver component) to be installed. It overlays the files of the old DSP with the files of the new DSP. **idinstall -u** requires that the module specified is currently installed.

idinstall -M *module-name* works whether or not the module is currently installed. It copies into the configuration directories any component files which are not yet installed or are newer than the installed versions. In any case, the files in the current directory are not removed.

When the **-a** or **-u** options are used, unless the **-e** option is used as well, **idinstall** attempts to verify that enough free disk space is available to start the reconfiguration process. This is done by calling the **idspace** command. **idinstall** will fail if there is not enough space and will exit with a non-zero return code.

idmkinit

idmkinit reconstructs **/etc/inittab** from the **Init** files in **/etc/conf/init.d**. The new **inittab** is normally placed in the **/etc/conf/cf.d** directory, although this can be changed through the **-o** option.

In the **sysinit** state during the next system reboot after a kernel reconfiguration, the **idmkinit** command is called automatically (by **idmkenv**) to establish the correct **/etc/inittab** for the running (newly-built) kernel. **idmkinit** is also called by **idbuild** when loadable kernel module configuration is requested. **idmkinit** can be executed as a user level command to test a modification of **inittab** before a DSP is actually built. It is also useful in installation scripts that do not reconfigure the kernel, but which need to create **inittab** entries. In this case, the **inittab** generated by **idmkinit** must be copied to **/etc/inittab**, and an **init q** command must be run for the new entry to take effect.

idmknod

idmknod reconstructs nodes (block and character special device files) in **/dev** and its subdirectories, based on the **Node** files for currently configured modules (those with at least one **Y** in their **System** files). Any nodes for devices with an '**r**' flag set in the *characteristics* fields of their **Master** file are left unchanged. All other nodes will be removed or created as needed to exactly match the configured **Node** files.

Any needed subdirectories are created automatically. Subdirectories which become empty as a result of node removal are removed as well.

All other files in the **/dev** directory tree are left unchanged, including symbolic links.

On the next system reboot after a kernel reconfiguration, in **sysinit** state, the **idmknod** command is run automatically (by **idmkenv**) to establish the correct representation of device nodes in the **/dev** directory tree for the running kernel. **idmknod** (with the **-M** option) is also called by **idbuild** when loadable kernel module configuration is requested. **idmknod** can be executed as a user level command to test modification of the **/dev** directory before a Driver Software Package

(DSP) is actually built. It is also useful in installation scripts that do not reconfigure the kernel, but which need to create **/dev** entries.

idspace

idspace checks whether sufficient free space exists to perform a kernel reconfiguration (see **idbuild**). By default, **idspace** checks the number of available disk blocks and inodes in three file systems: "**/**" and, if they exist, **/usr** and **/tmp**.

The default tests performed by **idspace** are

- Verify that the root file system ("**/**") has 400 blocks more than the size of the current **/stand/unix**. This verifies that a device driver being added to the current **/stand/unix** can be built and placed in the root file system. **idspace** also checks to ensure that 100 inodes exist in the root directory.
- Determine whether a **/usr** file system exists. If it does exist, **idspace** checks whether 400 free blocks and 100 inodes are available in the **/usr** file system. If the file system does not exist, **idspace** does not report an error, however, because files created in **/usr** by the reconfiguration process will be created in the parent root file system, and space requirements are covered by the **idspace** test of the root file system.
- Determine whether a **/tmp** file system exists. If it does exist, **idspace** checks whether 400 free blocks and 100 inodes are available in the **/tmp** file system. As with the test for the **/usr** file system, if the **/tmp** file system does not exist, **idspace** does not report an error, because files created in **/tmp** by the reconfiguration process will be created in the root file system, and space requirements are covered by the **idspace** test of the root file system.

Note that this function checks whether there is enough space to perform a reconfiguration, not whether there are enough free blocks and inodes to copy the DSP files from the installation media to the hard disk. To do this in your **postinstall** script, you should use the **df**(1M) command.

idtune

idtune sets or gets the value of an existing tunable parameter. **idtune** is called by a DSP installation or removal script; it can also be invoked directly as a user-level command. New tunable parameters must be installed using **idinstall**(1M) and a DSP **Mtune** file before they can be accessed using **idtune**.

Note that existing tunable parameter values must be modified using the **idtune** command.

The first form of the **idtune** command, with no options or with **-f** or **-m**, is used to change the value of a parameter. The tunable parameter to be changed is indicated by *parm*, and the desired value for the tunable parameter is *value*.

By default, if the parameter has already been tuned previously, you are asked to confirm the change with the message

> **Tunable Parameter** *parm* **is currently set to** *old_value* **in**
> **/etc/conf/cf.d/stune**
> **Is it OK to change it to** *value*? **(y/n)**

If you answer "**y**," the change is made. Otherwise, the tunable parameter will not be changed, and the following message is displayed

> *parm* **left at** *old_value*.

However, if you use the **-f** (force) option, the change is always made and no messages are reported.

If you use the **-m** (minimum) option, and the current value is greater than the desired value, no change is made and no messages are reported.

If you use the **-c** (current) option of the **idtune** command, the change applies to both **stune** and **stune.current**; otherwise, only the tunable parameter in **stune** is affected. **stune.current** contains the values currently being used by the running kernel; **stune** contains the values which will be used the next time the system is rebooted and the kernel rebuilt. Since any change made to the **stune.current** file will affect all the loadable kernel modules configured thereafter, it is very easy to introduce inconsistencies between the currently running kernel and the new loadable kernel modules. Therefore, you should be extremely careful when using the **-c** option.

If you are modifying system tunable parameters as part of a device driver or application add-on package, you may want to change parameter values without prompting the user for confirmation. Your DSP **postinstall** script could override the existing value using the **-f** or **-m** options. However, you must be careful not to invalidate a tunable parameter modified earlier by the user or another add-on package.

Any attempt to set a parameter to a value outside the valid minimum/maximum (as given in the **Mtune** file) range will be reported as an error, even when using the **-f** or **-m** options.

The UNIX system kernel must be rebuilt (using **idbuild**) and the system rebooted for any changes to tunable parameter values to take effect.

The Driver Software Package (DSP)

A DSP, from the users' perspective, is a software package they install on their system, usually so they can operate a piece of hardware, such as a network interface card or a disk drive, for example.

Users use the **pkgadd**(1) and **pkgrm**(1) command to install or remove the device drivers in DSPs. The **pkgadd** command installs a DSP from tape or floppy disk onto the system and initiates automatic procedures to reconfigure the kernel and to configure any loadable modules. The **pkgrm** command allows the user to select a package to delete from the system, removing the DSP and reconfiguring the kernel without the removed driver(s) or loadable module(s).

The **pkginfo**(1) command displays all the software packages the user has installed. DSPs are treated the same way as other SVR4.2 software packages. Device drivers pre-installed on the system by the Foundation Set are not displayed by this command.

What Is a DSP?

A Driver Software Package (DSP) consists of a driver object module, installation and removal scripts, and device-specific system configuration, initialization, and shutdown files. (Some of these files are optional and are not included in every DSP.)

The DSP is usually on a tape, or one or more floppy disks. To install the DSP, the user inserts the DSP media in the drive and runs the **pkgadd** command. This executes a script file in the DSP, which performs all the operations needed to copy all the object and configuration files from the installation media to the hard disk of the system. Then, the UNIX system kernel is reconfigured and built, and the user reboots the system to complete the installation.

What this means to you, as the device driver programmer, is that writing the driver is only part of the job. You also need to create the configuration files and write the installation and removal scripts. The DSP will also need to be tested, to make sure it can be installed and removed, as well as to ensure that it operates correctly when installed.

Once all the components have been created, copy them to the **/tmp** directory and use **pkgmk**(1) to create the DSP.

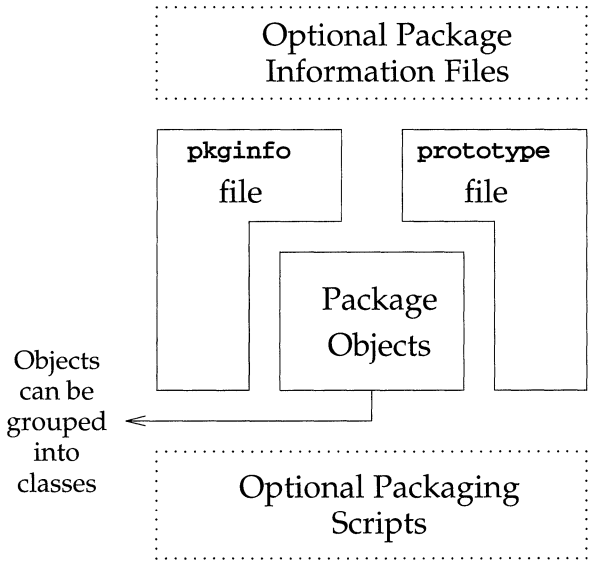
DSP Component Files

A software package is made up of a group of components that together create the software. These components naturally include the executables that comprise the software, but they also include at least two information files and can optionally include other information files and scripts.

As shown in Figure 3-1, a package's contents fall into three categories:

- required components (the **pkginfo** file, the **prototype** file, package objects)
- optional package information files
- optional packaging scripts

Figure 3-1: The Contents of a Package



Required Components

At the very least, a package must contain the following components:

- Package Objects

These are the objects that make up the software. They can be files (executable or data), directories, or named pipes. Objects can be manipulated in groups during installation by placing them into classes. For more information about classes, refer to the *UNIX Software Development Tools*.

- The **pkginfo** File

  The **pkginfo** file is a required package information file defining parameter values that describe a package. For example, this file defines values for the package abbreviation, the full package name, and the package architecture.

- The **prototype** File

  The **prototype** file is a required package information file that lists the contents of the package. There is one entry for each deliverable object and this entry consists of several fields of information describing the object. All package components, including the **pkginfo** file, must be listed in the **prototype** file.

The required package information files are described further in this chapter and on their respective manual pages.

Optional Package Information Files

There are four optional package information files that you can add to your package:

- The **compver** File

  Defines previous versions of the package that are compatible with this version.

- The **depend** File

  Defines any software dependencies associated with this package.

- The **space** File

  Defines disk space requirements for the target environment beyond that used by objects defined in the **prototype** file (for example, files that will be dynamically created at installation time).

- The **copyright** File

  Defines the text for a copyright message that will be printed on the terminal at the time of package installation or removal.

Every package information file used must have an entry in the **prototype** file. All of these files are described further in their respective manual pages.

Optional Installation Scripts

Your package can use three types of installation scripts, and although no scripts are required, they are recommended. Many of the tasks which had to be done manually in a installation script can be accomplished automatically by **pkgadd**. However, you may provide scripts with a DSP to perform customized actions. An installation script must be executable by **sh** (for example, a shell script or executable program). The three script types are the request script (solicits installer input), class action script (defines a set of actions to perform on a group of objects), and the procedure script (defines actions that will occur at particular points during installation).

Device Driver Packages

A DSP for a device driver will typically consist of the following components. Some are required, others are optional; this distinction is noted in Table 3-1.

- The driver module object file, **Driver.o**
- The configuration files for **Master**(4), **System**(4), **Mtune**(4), **Node**(4), **Rc**(4), **Sassign**(4), **Sd**(4), **Space.c**(4), and **Stubs.c**(4)
- **Modstub.o** for stub-loaded loadable modules.
- A **postinstall** script, which is used by the administrative command **pkgadd**(1M) to install the DSP
- A **preremove** script, which is used by the administrative command **pkgrm**(1M) to remove the DSP from the system
- A **prototype** file, which contains information about the contents of the DSP and which is used by **pkgmk** to create **pkgmap**, which contains details about the files that comprise the DSP
- A **pkginfo** file, describing characteristics of the DSP

The component files comprising the DSP are summarized in Table 3-1. In this table, the term *module-name* refers to a file or directory that takes its name from the name of the driver being installed. For the format of specific configuration files, you should refer to the appropriate Section 4 manual page.

Table 3-1: Components of Driver Software Package (DSP)

DSP Module	Purpose	File Affected in /etc/conf
`prototype`	Required OAM package prototype module	`none`
`pkginfo`	Required OAM package information module	`none`
`postinstall`	Optional script to install DSP package	`none`
`preremove`	Optional script to remove DSP package	`none`
`Driver.o`	Required driver object file to be configured into kernel	`pack.d/`*module-name*`/Driver.o`
`Master`	Required generic driver configuration data	`mdevice.d/`*module-name*
`Init`	Optional `inittab` entry data	`init.d/`*module-name*
`Mtune`	Optional tunable parameter definitions	`mtune.d/`*module-name*
`Node`	Optional `/dev` device node data	`node.d/`*module-name*
`Rc`	Optional system startup script	`rc.d/`*module-name*
`Sassign`	Optional system logical device name assignments	`sassign.d/`*module-name*
`Sd`	Optional system shutdown script	`sd.d/`*module-name*
`Space.c`	Optional driver data structure allocations and initializations	`pack.d/`*module-name*`/space.c`
`Stubs.c`	Optional stubs for symbols defined in a driver that will not be installed	`pack.d/`*module-name*`/stubs.c`
`Modstub.o`	Optional stub object file for loadable module	`pack.d/`*module-name*`/Modstub.o`

Overview of DSP Components

Following are each of the components that make up the typical DSP. Where possible, an example has been included to show you what the component might look like. Some are generic, while others are specific. The files **prototype**, **postinstall**, **pkginfo**, and **preremove** are only some of the packaging files that can be used in a software package. Note that very few DSPs include all of the possible components. It is also possible to have additional components, such as configuration or other script files.

For the more information on the files and file format described here, refer to the Section 4 manual pages. For more details about DSP Components, refer to the *UNIX Software Development Tools*.

prototype

The **prototype** file describes the DSP, listing where the files are to be installed and their characteristics. DSPs differ from a typical package, in that most of their component files are installed in **/tmp** for later processing by the **idinstall** command, which is called by the **postinstall** script.

Following is a generic example of a **prototype** file.

```
i pkginfo
i postinstall
i preremove

!default 644 root sys

d none  /tmp       ?         ?         ?
d none  /tmp/xyzzy

#
# These files are installed by the idinstall command in the postinstall script
#
v none    /tmp/xyzzy/Driver.o=/etc/conf/pack.d/xyzzy/Driver.o
v none    /tmp/xyzzy/Space.c=/etc/conf/pack.d/xyzzy/space.c
v none    /tmp/xyzzy/Master=/etc/conf/mdevice.d/xyzzy
v none    /tmp/xyzzy/System=/etc/conf/sdevice.d/xyzzy

#
# These files are installed by the postinstall shell script
#
v none    /tmp/loadmods=/newdrivers/xyzzy/loadmods
v none    /tmp/xyzzy/disk.cfg=/etc/conf/pack.d/xyzzy/disk.cfg

#
# This file is installed by the pkgadd command
#
f none    /usr/include/sys/xyzzy.h
```

For more information, refer to the **prototype**(4) manual page.

pkginfo

The **pkginfo** file describes various attributes for the DSP. For example, it identifies the name of the package as the value of the **PKG** variable.

Each entry in the **pkginfo** file is a line that establishes the value of a parameter in the following form.

PARAM="*value*"

There is no required order in which the parameters must be specified within the file.

Following is a generic example of a **pkginfo** file.

```
CATEGORY="system"
CLASSES="none"
NAME="XYZZY DRIVER PACKAGE"
PKG="xyzzy"
VENDOR="XYZZY Manufacturing Company"
VERSION="1"
```

For more information, refer to the **pkginfo**(4) manual page.

postinstall

postinstall is a shell script that performs the steps necessary to install the DSP on the system. **postinstall** does the following:

1. Changes directory to **/tmp**, where the DSP files were installed.
2. Executes **idinstall -a** and passes it the internal DSP name. This creates the needed directories and moves the DSP contents to the appropriate locations. If the **idinstall -a** fails, the package was already installed.
3. If the DSP has already been installed, **idinstall -u** command is used to update the package, using the files from the DSP. Another way to perform an update is to use the **-M** option, which updates only those installed files which are older than those in the DSP.
4. Runs the **idbuild** command without any options to create a new UNIX system kernel when the system is rebooted.
5. **removef** any **/tmp** files installed.

When writing a **postinstall** script, you should make liberal use of **echo** and **message** commands to tell the user what is going on. You should also make sure to exit with the appropriate return value based on a successful or unsuccessful installation.

Following is an example **postinstall** script.

```
do_install () {

  ${CONFBIN}/idinstall -a ${1} > ${ERR} 2>&1
  RET=$?
  if [ ${RET} != 0 ]
  then
    ${CONFBIN}/idinstall -u ${1} > ${ERR} 2>&1
    RET=$?
  fi

  if [ ${RET} != 0 ]
  then
    message "The installation cannot be completed due to an error in \
    the driver installation during the installation of the ${1} module \
    of the ${NAME}.  The file ${ERR} contains the errors."
      exit ${FAILURE}
  fi
  cp disk.cfg /etc/conf/pack.d/${1}
}

FAILURE=1 # fatal error
DRIVER=xyzzy
CONFDIR=/etc/conf
CONFBIN=${CONFDIR}/bin
ERR=/tmp/err.out

for MODULE in ${DRIVER}
do
  cd /tmp/${MODULE}
  do_install ${MODULE}
done

cat /tmp/loadmods >> /etc/loadmods

${CONFBIN}/idbuild >/dev/null 2>&1

installf -f $PKGINST

removef ${PKGINST} /tmp/loadmods /tmp/${DRIVER} >/dev/null 2>&1
removef -f ${PKGINST} >/dev/null 2>&1
```

preremove

The **preremove** shell script performs the steps to remove a DSP from a system. It does the following:

1. Uses **idcheck** to make sure the DSP to be removed exists on the system. If not, the script exits and displays an error message.

2. Runs **idinstall -d** and passes it the internal DSP name. This removes the DSP modules.
3. Invokes **idbuild** without any options to cause the kernel to be rebuilt when the system is rebooted.

Following is an example **preremove** script.

```
#ident    "@(#)install       43.7"

CONFDIR=/etc/conf
CONFBIN=${CONFDIR}/bin
DRIVER=xyzzy

for MODULE in ${DRIVER}
do
  ${CONFBIN}/idcheck -p ${MODULE}
  RES="$?"
  if
    [ "${RES}" -ne "100" -a "${RES}" -ne "0" ]
  then
    ${CONFBIN}/idinstall -d ${MODULE} 2>> /tmp/${MODULE}.err
  fi
done

${CONFBIN}/idbuild >/dev/null 2>&1

exit 0
```

Driver.o

A required component, the **Driver.o** component is the driver object module that is to be configured into the kernel. This object file should be compiled using the C Programming Language Utilities (CPLU), part of the Software Development Set.

Master

A required component, the **Master** file describes a kernel module for configuration into the system. The **System** file contains the configuration information for the individual kernel modules that are actually to be included in the next UNIX system kernel built [see **System**(4)].

When the **Master** component of a module's DSP is installed, **idinstall** stores the module's **Master** file information in **/etc/conf/mdevice.d/***module-name*, where the file *module-name* is the name of the driver module being installed.

DSP package scripts should never access **Master** files directly; they should use the **idinstall** and **idcheck** commands instead.

Master files contain lines of the form:

> **$version** *version-number*
> **$entry** *entry-point-list*
> **$depend** *module-name-list*
> **$modtype** *loadable-module-type-name*
> *module-name prefix characteristics order bmaj cmaj*

Blank lines and lines beginning with '**#**' or '*****' are considered comments and are ignored.

Following is an example **Master** file for the **st01** tape driver.

```
#ident  "@(#)/etc/conf/mdevice.d/st01.sl 1.2 SVR4.2 02/04/92 34022 USL"
#ident  "$Header: $"
$version 1
$entry init open close read write ioctl
st01    st01    kocr    0       0       22
```

For complete information about the **Master** file format, refer to the **Master**(4) manual page.

System

A required component, the **System** file contains information needed to incorporate a particular kernel module into the next UNIX system configuration. General configuration information about the module type is described in the **Master** file. When the **System** component of a DSP is installed, **idinstall** stores the module's **System** file information in **/etc/conf/sdevice.d/***module-name*, where the file *module-name* is the name of the module being installed.

DSP package scripts should never access **Master** files directly; they should use the **idinstall** and **idcheck** commands instead.

System files contain lines of the form:

> **$version** *version-number*
> **$loadable** *module-name*
> *module-name configure unit ipl itype ivec sioa eioa scma ecma dmachan*

Blank lines and lines beginning with '**#**' or '*****' are considered comments and are ignored.

Following is an example **System** file for the **st01** tape driver.

```
*ident  "@(#)/etc/conf/sdevice.d/st01.sl 1.4 SVR4.2 04/09/92 5743 USL"
*ident  "$Header: $"
$version 1
$loadable st01
st01    Y       0       0       0       0       0       0       0       0       -1
```

For complete information about the **System** file format, refer to the **System**(4) manual page.

Init

An optional component, the **Init** file contains information used by the **idmkinit** command to construct a module's **/etc/inittab** entry. When the **Init** component of a module's DSP is installed, **idinstall** stores the module's **Init** file information in **/etc/conf/init.d/***module-name*, where the file *module-name* is the name of the module being installed.

DSP package scripts should never access **Init** files directly; they should use the **idinstall** command instead. **Init** files contain line consisting of one of the following three forms:

```
action:process
rstate:action:process
id:rstate:action:process
```

All fields are positional and must be separated by colons. Blank lines and line beginning with '**#**' or '*****' are considered comments and are ignored.

Lines of the first form should be used for most entries. When presented with a line of this form, **idmkinit**:

1. Copies the *action* and *process* field to the **inittab** entry.
2. Generates a valid *id* field value (called a 'tag') and prepends it to the entry.
3. Generates an *rstate* field with a value of 2, and adds it to the entry, following the *id* field.

Lines of the second form should be used when an **rstate** value other than **2** must be specified. When presented with a line of this form, **idmkinit** generates only the **id** field value and prepends it to the entry.

Lines of the third form should be used with caution. When presented with a line of this form, **idmkinit** copies the entry to the **inittab** file verbatim. It is recommended that DSPs avoid specifying lines of this form because, if more than one DSP or add-on application specifies the same *id* field, **idmkinit** will create multiple inittab entries containing this **id** value. When the **/etc/init** program attempts to process the **inittab** entries with the same *id*, it will fail with an error condition.

Note that **idmkinit** determines which of the three forms is being used by searching each line for a valid *action* keyword. Valid *action* values are:

```
boot
bootwait
initdefault
off
once
ondemand
powerfail
powerwait
respawn
systinit
wait
```

For complete information about the **Init** file format, refer to the **Init**(4) manual page.

Mtune

An optional component, the **Mtune** file contains definitions of tunable parameters, including default values, for a kernel module type.

When the **Mtune** component of a DSP is installed, **idinstall** stores the module's **Mtune** file information in **/etc/conf/mtune.d/***module-name*, where the file *module-name* is the name of the module being installed.

DSP package scripts should never access **Mtune** files directly; they should use the **idinstall** and **idtune** commands instead.

Mtune files contain lines of the form:

parameter-name default-value minimum-value maximum-value

All fields are positional and must be separated by white space. Blank lines and lines beginning with '**#**' or '*****' are considered comments and are ignored.

Following is an example **Mtune** file for **kmacct**, KMA (Kernel Memory Allocation) Accounting.

```
#ident  "@(#)/etc/conf/mtune.d/kmacct.sl 1.1 SVR4.2 10/05/91 1640 USL"
#ident  "$Header: $"

* KMACCT Tunables ------------------------------

* KMARRAY is the number of entries in the symbol table

KMARRAY 150     50      500

* SDEPTH is the depth of stack to trace back (no larger than MAXDEPTH
* from sys/kmacct.h)

SDEPTH  5       3       10

* NKMABUF is the number of buffer headers to allocate (one for each
* buffer that has been allocated and not yet returned).

NKMABUF 1000    100     10000
```

For complete information about the **Mtune** file format, refer to the **Mtune**(4) manual page.

Node

An optional component, the **Node** file contains definitions used by the **idmknod**(1M) command to create the device nodes (block and character special files) associated with a device driver module. When the **Node** component of a module's DSP is installed, **idinstall** stores the driver's **Node** file information in **/etc/conf/node.d/***module-name*, where *module-name* is the name of the driver being installed.

DSP package scripts should never access **Node** files directly; they should use the **idinstall** command instead.

Each device node for the driver is specified on a separate line of the form:

module-name node-name type minor user group permissions

All fields are positional and must be separated by white space. The first four fields are required; the last three fields are optional. Blank lines and lines beginning with '**#**' or '*****' are considered comments and are ignored.

Following is an example **Node** file for **log**, the UNIX system event logger.

```
#ident  "@(#)/etc/conf/node.d/log.sl 1.1 SVR4.2 10/05/91 58071 USL"
#ident  "$Header: $"

log     log     c       5       0       0       444
log     conslog c       0       0       0       222
```

Rc

An optional component, the **Rc** file is an optional file that executes when the system is booted to initialize an installed kernel module. Normally, this is a shell script [see **sh**(1)].

When the **Rc** component of a module's DSP is installed, **idinstall** stores the module's **Rc** file in **/etc/conf/rc.d/***module-name*, where *module-name* is the name of the module being installed.

DSP package scripts should never access **Rc** files directly; they should use the **idinstall** command instead.

The contents of the **/etc/conf/rc.d** directory are linked to **/etc/idrc.d** whenever a new configuration of the kernel is first booted. On this initial reboot, and on all subsequent reboots, the module's **Rc** file is invoked upon entering **init** level 2 [see **init**(1M)].

Following is an example **Rc** file for **pts**:

```
if [ -c /dev/pts000 ]
then
exit
fi
cd /dev/pts
for i in *
do
NUM='echo $i | awk '{printf("%.3d",$1)}''
ln $i /dev/pts${NUM} >> /dev/null 2>&1
done
```

Sassign

An optional component, the **Sassign** file give system administrators the ability to assign specific actual devices to logical device names used by the kernel. One example is **rootdev**, which is the device that contains the root file system. At present, **Sassign** supports only block devices.

If the system administrator wants to assign a different actual device to perform a function, the administrator remaps the logical device name for that function to specify configured device in the **Sassign** file. Note that the kernel must be rebuilt and rebooted for the new assignment to take effect.

Each logical device name in the **Sassign** file is specified (in **/etc/conf/sassign**) on a separate line of the form:

device-variable-prefix device-module-name minor node-name

All fields are positional and must be separated by white space. Blank lines and lines beginning with '**#**' or '*****' are considered comments and are ignored. The *node-name* field is applicable to swap devices only.

Following is an example **Sassign** file:

```
#ident  "@(#)/etc/conf/sassign.d/kernel.sl 1.1 SVR4.2 10/05/91 17775 USL"
#ident  "$Header: $"

* Device variable assignments for the base kernel.
swap    sd01    2       /dev/swap
dump    sd01    2
root    sd01    1
```

For complete information about the **Sassign** file format, refer to the **Sassign**(4) manual page.

Sd

An optional component, **Sd** is a file that executes when the system is shut down to perform any cleanup required for an installed kernel module. Normally, this is a shell script [see **sh**(1)].

When the **Sd** component of a module's DSP is installed, **idinstall** stores the module's **Sd** file in **/etc/conf/sd.d/***module-name*, where *module-name* is the name of the module being installed.

DSP package scripts should never access **Sd** files directly; they should use the **idinstall** command instead.

The contents of the **/etc/conf/sd.d** directory are linked to **etc/idsd.d** whenever a new configuration of the kernel is first booted. On this initial reboot, and on all subsequent reboots, the module's **Sd** file is invoked upon entering **init** level 0, 5, or 6 [see **init**(1M)].

Space.c

An optional component, the **Space.c** file contains storage allocations and initializations of data structures associated with a kernel module, when the size or initial value of the data structures depend on configurable parameters, such as the number of subdevices configured for a particular device or tunable parameter. For example, the **Space.c** file gives a driver the ability to allocate storage only for the subdevices being configured, by referencing symbolic constants defined in the **config.h** file. The **config.h** file is a temporary file created in **/etc/conf/cf.d** during the system reconfiguration process.

When the **Space.c** component of a module's DSP is installed, **idinstall** stores the module's **Space.c** file in **/etc/conf/pack.d/***module-name***/space.c**, where *module-name* is the name of the module being installed.

DSP package scripts should never access **Space.c** files directly; they should use the **idinstall** command instead.

Following is an example **Space.c** file for the **st01** tape driver.

```
#ident  "@(#)/etc/conf/pack.d/st01/space.c.sl 1.3 SVR4.2 06/18/92 59307 USL"
#ident  "$Header: $"

#include <sys/types.h>
#include <sys/scsi.h>
#include <sys/conf.h>
#include <sys/sdi_edt.h>
#include <sys/sdi.h>
#include "config.h"


struct dev_spec *st01_dev_spec[] = {
        0
};

struct dev_cfg ST01_dev_cfg[] = {
{       SDI_CLAIM|SDI_ADD, 0xffff, 0xff, 0xff, ID_TAPE, 0, ""   },
};

int ST01_dev_cfg_size = sizeof(ST01_dev_cfg)/sizeof(struct dev_cfg);

int     St01_cmajor = ST01_CMAJOR_0;    /* Character major number       */

int     St01_jobs = 20;         /* Allocation per LU device     */

int     St01_reserve = 1;       /* Flag for reserving tape on open */
```

For complete information about the **Space.c** file format, refer to the **Space.c**(4) manual page.

Stubs.c

An optional component, a **Stubs.c** file is a C language source file that can be installed and compiled into the system as a "placeholder" for a kernel module that will not be installed in the system at this time. Its purpose is to enable the kernel to resolve references to the absent module's symbols.

A module's **Stubs.c** file contains function name and variable definition stubs for symbols defined in the module that can be referenced by other kernel modules being configured into the system. At compile time, the definitions in the **Stubs.c** file give the kernel the ability to resolve references made to the absent module's symbols.

When the **Stubs.c** component of a module's DSP is installed, **idinstall** stores the module's **Stubs.c** file information in **/etc/conf/pack.d/***module-name***/stubs.c**, where *module-name* is the name of the module being installed.

The **Stubs.c** file needs to be handled differently in **preremove** scripts if it should be kept even after the DSP is removed. This is done by using **idinstall -g**.

DSP package scripts should never access **Stubs.c** files directly; they should use the **idinstall** command instead.

Following is an example **Stubs.c** file for **log**, the UNIX system event logger.

```
#ident  "@(#)/etc/conf/pack.d/log/stubs.c.sl 1.2 SVR4.2 01/31/92 21292 USL"

int
strlog()
{
return(0);
}

int
conslog_set()
{
return (0);
}
```

For complete information about the **Stubs.c** file format, refer to the **Stubs.c**(4) manual page.

Modstub.o

An optional component, the **Modstub.o** file is an object module for stub-loaded loadable modules. This object file, like the **Driver.o** component, should be compiled using the C Programming Language Utilities (CPLU), part of the Software Development Set.

Packaging the Driver

For complete information on the UNIX system packaging tools, refer to the *UNIX Software Development Tools* and the applicable Section 4 manual pages for the DSP component files. However, following is a brief summary of what is required to create a DSP, presented here to provide a better context for understanding.

To help create the **prototype** file, the **pkgproto** command can take command line arguments to scan a development directory structure and generate the **prototype** file. The **prototype** file generated by **pkgproto**, however, lists the components in the directory structure used on the development machine; therefore, it will be installed into the same directories on the user's system.

To package a driver, put all of the component files into the directories specified in the **prototype** file and use the **pkgmk** command. **pkgmk** uses the **prototype** and **pkginfo** files to create a file called **pkgmap**(4) and creates the DSP.

The **pkgtrans**(1) command copies a DSP to the installation media, either tape or floppy disks.

Each DSP must have two "names." One is the "external name" that the user sees when the package is installed. The second is an "internal" name that the kernel uses to identify the device.

The DSP's **prototype** file should install the component files as class "volatile" in the **/tmp** directory. Then, the **postinstall** script, when executed, should **cd** to that directory before executing **idinstall** to add the DSP to the system.

Typical DSP Installation and Removal Scenarios

Installing a DSP

A user installing a DSP usually will find the process very simple. From the user perspective, a typical installation proceeds as follows:

1. The user runs the **pkgadd** command with the **-d** *device* option, where *device* specifies the floppy disk or tape drive where the DSP is to be installed from. For example, *device* could be **disk1**.
2. A prompt asks the user to insert the floppy disk or mount the tape.
3. A second prompt appears, asking the user which DSP is to be installed or whether to install all DSPs on the installation media.
4. The DSP package is installed, a process which may take several minutes or longer, depending on the DSP. This process usually does not require any particular user intervention.
5. A message is displayed signaling success or failure of the installation.
6. A prompt asks the user whether another DSP is to be installed. If so, this process is repeated.
7. When all desired DSPs have been installed, a message is displayed, telling the user to reboot the system to complete the DSP installation process.

Removing a DSP

As shown above, the installation process is relatively simple and straightforward from the user's viewpoint. Removing a package is even easier.

1. The user executes the **pkgrm** command.
2. A prompt asks the user which DSP to remove.
3. The **preremove** script deletes all the files and commands associated with the DSP, calling the **idinstall -d** command.
4. A prompt is displayed, instructing the user to reboot the system to complete the DSP removal.

DSP Commands and Procedures

The three most important idtools commands for DSPs are **idcheck**, **idinstall**, and **idbuild**, and every **postinstall** and **preremove** script should use all three of them.

For example, at bare minimum, the **postinstall** script will call **idcheck** to see whether the DSP has already been installed. Then, the script runs **idinstall**, either with the **-a** option to install the DSP or with the **-u** option to update an existing DSP. Finally, the **postinstall** calls **idbuild** to build a new UNIX system base kernel and/or configure loadable modules.

The **preremove** script, used to remove a DSP from the system, also uses **idcheck** to see whether the DSP exists (there is no point in attempting to remove a DSP that is not there). Then, the **idinstall** command is run using the **-d** option; this deletes the component files and configuration file entries relating to the DSP. (Sometimes the **Stubs.c** needs to be kept; refer to **idinstall**(1M) to see how to do this.) Lastly, the script calls the **idbuild** command to build a new kernel, without the DSP, and/or to remove configuration data for loadable modules.

Of the abovementioned commands, **idinstall** is the one that performs the widest range of functions. It does not just install DSPs, but it can also update or remove existing DSPs on a system. A DSP installation or removal script calls **idinstall** to add (**-a**), delete (**-d**), update (**-**), or get (**-g** or **-G**) device driver and kernel module configuration data. It can also be run from a kernel source makefile to make (**-M**) configuration data.

idinstall expects to find DSP component files in the current directory, which, for DSP installation purposes, should be **/tmp**. When the components are installed or updated with the **-a** or **-u** option, they are copied into the subdirectories of the **/etc/conf** directory. Then, the files are deleted from the current directory, unless the **-k** option is used, which tells **idinstall** to keep the files.

Checking the System Configuration

The **idcheck** command returns selected information about the system configuration. In DSP scripts, it can be used to determine whether a particular device driver has already installed, and to verify that a particular interrupt vector, I/O address, or other selected parameter is, in fact, available for use.

The options available for the **idcheck** command enable you to select which item to check for, but it is the **-p** *module-name* option which checks for the existence of a particular DSP's modules. **idcheck** returns a numeric value depending on which components it finds, or **0** if no components are found.

Other options check for conflicting devices, interrupt vectors, DMA channels, address ranges, and other details.

For complete information about the **idcheck** command, refer to the **idcheck**(1M) manual page.

Installing a DSP

To install a DSP, the **postinstall** script needs to call the **idinstall** command with the **-a** option. An example command for installing a DSP follows:

idinstall -a *module-name*

In this example, *module-name* represents the name of the DSP to be installed. Unless the **-e** option is also specified, **idinstall** performs a check to see whether there is enough free disk space to start the configuration process, calling **idspace** to do this. Note that this check for available disk space is different from the one you should perform in the **postinstall** script. In the script, you should use **df**(1M) to check whether there are enough free blocks and inodes to copy the component files from the installation media to the **/tmp** directory.

The **idinstall** command requires and installs the DSP **Driver.o**, **Master**, and **System** components, and, if present, installs the **Init**, **Mtune**, **Node**, **Rc**, **Sassign**, **Sd**, **Space.c**, and **Stubs.c** components, all of which must be in the current directory.

The **Driver.o**, **Space.c**, and **Stubs.c** components are moved to a directory named **/etc/conf/pack.d/***module-name*. The remaining files are stored in directories under **/etc/conf**, which are organized by component type, in files named *module-name*. For example, the **Node** file would be moved to **/etc/conf/node.d/***module-name*.

For complete information about the **idinstall** command, refer to the **idinstall**(1M) manual page.

Updating a DSP

If a check for the existence of the DSP (using **idcheck**) turns up positive, a **postinstall** script should use the **idinstall** update option. This is assuming that it makes sense to update the DSP, and in any event, you should require a positive verification, or at least give the user the option of aborting, before updating an existing DSP.

The update can either completely overwrite the existing DSP files on the system, or overwrite them selectively, based on whether each file in the DSP is newer than the one on the system.

The following examples update a DSP:

idinstall -u *module-name*

or

idinstall -M *module-name*

The first command overwrites all the files of the original DSP with files of the new DSP, requiring that the *module-name* specified is currently installed. The **idinstall -u** *module-name* command requires that the module specified is currently installed.

The second variation, **idinstall -M** *module-name*, works whether or not the DSP is currently installed. It copies into the appropriate configuration directories any component files which are not yet installed or are newer than the installed versions. In any case, with **idinstall -M**, the files in the current directory are not removed.

For complete information about the **idinstall** command, refer to the **idinstall**(1M) manual page.

Removing a DSP

To remove a DSP from the system, a **preremove** script needs to call the **idinstall** command with the **-d** option. An example command follows.

idinstall -d *module-name*

In the example, *module-name* is the name of the DSP to be removed. Once executed, all files and commands associated with the DSP are removed. A reboot is required to reconfigure the kernel once the DSP has been removed.

Building a New Kernel

A new kernel needs to be built when installing or removing a DSP, after all of the DSP component modules (for example, **Master**, **System**, **Init**, and so on) have been installed or removed from the appropriate locations. It is usually a good idea to reboot after a DSP update, as well. The **idbuild** command builds a UNIX system base kernel and/or configures loadable kernel modules using the current system configuration in *$ROOT*/*$MACH*/**etc/conf**.

Base kernel rebuilds are usually needed after a statically linked kernel module is installed, when any static module is removed, or when system tunable parameters are modified. If you execute **idbuild** without any options and if the environment variable *$ROOT* is null or "/", a flag is set and the kernel rebuild is deferred to next system reboot.

When adding or removing a DSP through the **postinstall** or **preremove** scripts, you may want to use the **idbuild -B** command to build a new kernel immediately, although if installing several packages at once, you probably will not want to rebuild the kernel until after all the DSPs are installed. Then, the system is rebooted using the new UNIX system kernel in **/stand/unix**, with the old kernel saved as **unix.old** if there is enough disk space available.

When loadable modules are to be added, you use the **-M** *module-name* option, repeating the option on the command line as many times as needed to configure all the loadable modules.

The **-B** and **-M** options can be used on the same command line.

Building a UNIX system kernel consists of three steps.

1. Generate configuration tables and symbols, and module lists from the configuration data files.
2. Compile configuration-dependent files, and then link these together with all of the configured kernel and device driver object modules.
3. If the loadable kernel module feature or a kernel debugger is enabled, attach the kernel symbol table information to the kernel.

The kernel is, by default, placed in *$ROOT*/*$MACH*/**etc/conf/cf.d/unix**.

If the kernel build is successful and *$ROOT* is null or "/", **idbuild** sets a flag to instruct the system shutdown/reboot sequence to replace the standard kernel in **/stand/unix** with the new kernel. Then, another flag will be set to cause the environment (device special files, **/etc/inittab**, and so on) to be reconfigured accordingly.

If one or more loadable kernel modules are specified with the **-M** option, **idbuild** will configure only the specified loadable kernel modules and put them into the *$ROOT*/*$MACH*/**etc/conf/mod.d** directory. Otherwise a UNIX system base kernel is rebuilt with all the loadable modules reconfigured into the *$ROOT*/*$MACH*/**etc/conf/modnew.d** directory, which will be changed to **/etc/conf/mod.d** at the next system reboot, if *$ROOT* is null or "**/**" [see **modadmin**(1M)].

When loadable kernel modules are configured with the **-M** option, **idbuild** also creates the necessary nodes in the **/dev** directory, adding and activating **/etc/inittab** entries if any **Init** file is associated with the modules, and registering the modules to the running kernel [see **idmodreg**(1M)]. This makes them available for dynamic loading without requiring a system reboot.

For complete information about the **idbuild** command, refer to the **idbuild**(1M) manual page.

Rebooting the System with the New Kernel

After adding or removing DSPs, the system needs to be rebooted for the changes to take effect. Once rebooted, modules configured for static installation with the kernel are initialized, and modules configured for dynamic loading are made available.

Emergency Recovery (New Kernel Will Not Boot)

It is possible that the kernel will fail to boot if your driver contains a serious bug. This can be due to a **panic** call that you put in your driver or some other system problem. If this happens, you should reset the system and boot the original kernel, which would be saved by **idbuild** in **/stand/unix.old** if there was enough disk space available to make the copy. To do this, reset the system, and when you see the message

```
Booting UNIX System ...
```

quickly press the console keyboard spacebar to interrupt the default boot. When the boot prompt appears, type "**/stand/unix.old**" or whatever name you may have used for a backup copy of the kernel.

If you do not have a working backup copy of the kernel or some other disaster has occurred, and you cannot recover gracefully, you will need to follow the procedure listed below to put a bootable **/stand/unix** back on the hard disk.

1. Boot the system from the first boot floppy.
2. When the system prompts you to do so, insert the second boot floppy.
3. When prompted, select the non-destructive installation, as if you needed to overlay the system.
4. After loading the third boot floppy, you will be prompted to select Automatic or Custom installation. Press DEL to get the interruption screen. Press DEL again to get a shell prompt.
5. At this time,the hard disk file system tree is mounted on **/mnt**. You can either build a new kernel, or mount the first floppy and copy the boot kernel to **/mnt/stand/unix**.
6. Press the RESET button, or power down and then back up again.

The system should now boot normally with a standard foundation kernel. Your new driver and any other drivers you had installed on your system will not be included in the UNIX kernel, even though they may appear in the **pkginfo** output. To fix this, remove your driver and execute **idbuild**. If that fails, remove and reinstall all of the packages.

This procedure can also be useful if other system files are damaged inadvertently while debugging your driver. There are several reasons why your system may fail to boot properly or not let you log in after it has booted. For example, a corrupted password or **inittab** could prevent console logins.

The contents of the three boot floppy disks are copied to a temporary root file system on the hard disk, including a default **/etc/passwd**, **/etc/init**, **/etc/inittab**, and other critical files. When using the previously listed procedures, you can copy the default files from the temporary root file system mounted on "**/**" to the hard disk root, currently mounted as **/mnt**. Obviously, user logins you have added to **/etc/passwd** and other system changes you have made since installing the original base system will be lost when you overwrite the corrupted file with the floppy disk default file. A better solution is to make regular, scheduled backups of your hard disk, especially for critical system configuration files.

Documenting Your Driver Installation

If you are developing a DSP to be installed by users who may not be familiar with the implications of reconfiguration, some words of caution may be worthwhile.

- Although experience has shown little difficulty in installing and removing a variety of device drivers, there is the possibility that you may have difficulty booting the system. The cause of this probably would be due to some fault in the added driver. If this occurs, you may have to restore the UNIX system kernel from the saved version (created by **`idbuild`** upon system reconfiguration, assuming there is enough disk space for the backup).
- Since a reconfiguration often ends with a system reboot, it is not advisable for other users to be logged in to the system through a remote terminal.
- Users should not press DEL or RESET, power down the system, or in any way try to interrupt an installation. Although interruption protection is built into the idtools scheme, total protection against a reboot during an installation can never be completely foolproof.
- Use the **`df`** command in your script or advise your users to run **`df`** to determine the free disk space before doing the installation. If there is not enough space to install the DSP, tell the user how much space needs to be freed up. If you require the users to check for themselves, tell them how many free blocks are needed to install the DSP.
- Similarly, if your script exits because **`idspace`** has revealed that there is not enough space to reconfigure the kernel, tell the user how many blocks are needed.
- Advise the user not to have any background processes running that will be adversely affected by a system reboot or consume free disk space while a reconfiguration is underway. For example, avoid running **`uucp`** during an installation.

Device Driver Tuning

The **Mtune** files contain kernel tunable parameters which can profoundly affect system performance. Occasionally an add-on device driver or kernel software module may require you to modify an existing parameter or define a new tunable parameter that is accessible by other add-on drivers. Note, however, that not all drivers have or require tunable parameters.

When the **Mtune** component of a module's DSP is installed, **idinstall**(1M) stores the module's **Mtune** file information in **/etc/conf/mtune.d/***module-name*, where the file *module-name* is the name of the module being installed. An **Mtune** file defines a default value along with a minimum and maximum value for each tunable parameter of a particular driver module.

NOTE

Package scripts should never access **/etc/conf/mtune.d** files directly; only the **idinstall** and **idtune** commands should be used.

Modifying a Kernel Parameter

The **idtune** command is used to modify system-tunable parameter entries in the **stune** file, from the default value in the **Mtune** file. Not every system-tunable parameter is contained in the **stune** file; only those that are to be set to a value other than the system default need be entered there. Therefore, if the driver package you are building requires modifying a parameter value, you should use the **idtune** command only. Never modify the **stune** file directly.

The **idtune** command takes individual system parameters, verifies that the new value is within the upper and lower bounds specified in **Mtune**, searches the **stune** file, and modifies an existing value or adds the parameter to **stune** if not defined.

The **stune** file (located in **/etc/conf/cf.d/stune**) file contains tunable parameters for the kernel modules to be configured into the next system to be built (see **idbuild**). The parameter settings in the **stune** file are used to override the default values specified in the **Mtune** file.

The contents of the **stune** file will only affect the next kernel rebuild. Once the new kernel has been installed to **/stand/unix** and booted, the **stune** file is copied to **stune.current**. Any change made to the **stune.current** file using the **idtune** command with the **-c** option will affect all the loadable kernel modules subsequently configured into the running system.

The **stune** and **stune.current** files contain one line for each parameter to be set. Each line contains two positional fields separated by white space. *parameter-name new-value*

Package scripts should never access **/etc/conf/cf.d/stune** or **/etc/conf/cf.d/stune.current** files directly; only the **idtune** command should be used.

Device Driver Configuration

When installing a device driver, you can specify whether the driver is to be static—that is, configured into the base kernel—or to be dynamically loadable.

A dynamically loadable module can be loaded upon demand or automatically whenever the system receives a request requiring the module. Similarly, a loadable module can be unloaded upon demand, or it can be configured to be unloaded after a certain amount of time has elapsed since it was last accessed.

To configure a module to be loadable (this is usually done in the **postinstall** script of a DSP):

```
idbuild -M "module-name"
```

To statically link a module into the base kernel, comment out the "**$loadable**" line in its **System** file, then use **idbuild** and reboot.

```
idbuild -S
```

Refer to the **idbuild**(1M) manual page for more information.

Loadable Module Administration

modadmin is the administrative command for loadable kernel modules. It performs the following functions.

- Load a loadable module into a running system
- Unload a loadable module from a running system
- Display the status of a loadable module(s) currently loaded
- Modify the loadable modules search path

The loadable modules feature lets you add a module to a running system without rebooting the system or rebuilding the kernel. When the module is no longer needed, this feature also lets you dynamically remove the module, thereby freeing system resources for other use.

For more information about loadable module administration and how it impacts device driver programming, refer to Chapter 2, "Loadable Modules".

4 Driver Testing and Debugging

Introduction

Testing a device driver consists of installing the driver on a working system and attempting to try all of its functions under a variety of operating conditions. Debugging a driver is largely a process of analyzing the code to determine what could have caused a given problem. The UNIX system includes some tools that may help, but because the driver operates at the kernel level, these tools can only provide limited information.

This chapter describes the tools that are available for testing and debugging the installed driver and how to use them. This chapter also discusses some of the common errors in drivers and some of the symptoms that might identify each.

During the first phases of testing, remember that your driver code is probably not perfect, and that bugs in the driver code may panic or damage the system, even parts of the system that may seem unrelated to your driver. Testing should be done when no other users are on the system and all production data files are backed up. Alternatively, testing could be performed on a restricted-use system set up specifically for the purpose of testing drivers.

You should test the functionality of the driver as you write it. If you are changing code from another driver, it is useful to install and test the driver after you have modified the initialization routines and the **`read/write`** or **`strategy`** routines. This testing could involve writing a short program that only reads and writes to the device to ensure that you can get into the device. When all the routines for the driver are written, you should install the hardware and perform full functionality testing.

The UNIX system provides tools to help you, such as **`crash`**(1M), which is used either for a post-mortem analysis after a system failure or for interactive monitoring of the driver.

Preparing a Driver for Debugging

The process of testing driver functionality is piecemeal: you have to take small pieces of your driver and test them individually, building up to the implementation of your complete driver.

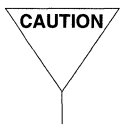
Driver routines should be written and debugged in the following order:

1. **init**(D2X), **start**(D2X)
2. **open**(D2X), **close**(D2X)
3. **int**, **intr**, or **rint/xint** interrupt routines
4. **ioctl**(D2X), **read**(D2X), **write**(D2X) and/or **strategy**(D2X) and **print**(D2X)

When the driver seems to be functioning properly under normal conditions, begin testing the error logs by provoking failures. For instance, take a tape or disk offline while a read/write operation is going.

After you are comfortable that both the hardware and software behaves as it should during error situations, it is time to concentrate on formal performance testing.

General Guidelines



Before trying to install or debug the driver, back up all files in your file system(s). Drivers can cause serious problems with disk sanity should an unanticipated problem occur.

Compile your driver and produce an up-to-date listing and an object file. The following conventions must be observed:

- Ensure that all your **cmn_err**(D3X) calls direct output to at least the **putbuf** memory array. (**putbuf** defaults to a maximum size of 10,000 bytes.)
- Compile your driver without the optimizer, with the **-g** option enabled.
- Use the **pr -n**(1) command to produce a listing of the source code with line numbers. Alternatively, **list**(1) can be used to pull line number information out of the driver object file.

- Use **dis**(1) to produce a disassembly listing. This is useful to have on hand, even though you get the same information using the **crash dis** command.
- Use **list**(1) to produce a listing that correlates the line numbers in the disassembly listing back to original source file.

Using the instructions described earlier in this chapter, install your driver. If the UNIX system does not come up, divide your driver into separate sections and install each part separately until you find the problem. Fix the problem and install the driver.

After the driver is installed, use **idbuild**(1M) to create the **/stand/unix** file.

In single-user mode, run **nm**(1) on **/stand/unix** (with the **-nef** options) to create a name list for the entire kernel. All addressing is virtual. The name list gives the starting locations (routine names and starting addresses) of the instructions and variables.

Putting Debug Statements in a Driver

Use the **cmn_err**(D3X) function to put debugging comments in the driver code; when the driver executes, you can use these to tell what part of the driver is executing. The **cmn_err** function is similar to the **printf**(3S) system call but it executes from inside the kernel.

cmn_err statements for debugging should be written to the **putbuf** where they can be viewed using **crash**. Because they are written by the kernel, they cannot be redirected to a file or to a remote terminal. You can also write **cmn_err** statements to the console, but massive amounts of statements to the console will severely slow system speed.

Calculations and **cmn_err** statements that are for debugging and other testing should be coded within conditional compiler statements in the driver. This saves you the task of removing extraneous code when you release the driver for production, and makes that debugging code readily available should you need to troubleshoot the driver after it is in the field. You can provide separate code for different types of testing to which the driver will be subjected. For instance, you might use **TEST** for functionality testing, **PERFON** for minimal performance testing, and **FULLPERF** for full performance monitoring. Each of the testing options is then defined in the code as either **0** (turned off) or **1** (turned on), as illustrated below.

```
/* TEST = 1 for functionality testing
*/
#define TEST       1
/*
* PERFON = 1 for minimal performance monitoring
*/
#define PERFON     0
/*
* FULLPERF = 1 for full performance monitoring
*/
#define FULLPERF   1
```

Note that minimal performance monitoring is turned off, which is appropriate because full performance monitoring is turned on.

Debug code is then enclosed within **#if TEST** and **#endif**. When the code is compiled with the **-DTEST** option, the test code will execute.

The testing procedure can be refined further by using flags within the conditionally-compiled code. Then, when **TEST** is turned on, you can specify the exact sort of testing without recompiling and reinstalling the driver. The flags should use the driver prefix. For instance, the following code sets three flags for testing the **int**(D2X) interrupt routine, the **strategy**(D2X) routine, and driver performance:

```
#if TEST
int xx_intpr, xx_stratpr, xx_perfpr;
#endif
```

The flags reside as the first words in the **.bss** section of the driver code. To turn on one or more flags

- Get the start address of **.bss** from the namelist with a command similar to

  ```
  nm -x /stand/unix | egrep 'xx_intpr|xx_stratpr|xx_perfpr'
  ```

- Write a short program that prompts you for the address of the flag(s) you want turned on, then specifies location in memory

Installing a Driver for Testing

Many of the steps that follow require you to modify files and directories owned by **root**. You must therefore be logged in as **root** or execute with the appropriate privileges to develop and debug device drivers.

1. First of all, it would be a good idea to make a copy of your current UNIX operating system kernel before reconfiguring the system. The backup is made automatically by the **idbuild** command saving the kernel as **/stand/unix.old** (if there is enough disk space), but it is still a good idea to have a 'pre-driver test' backup kernel, because the second and subsequent executions of **idbuild** will overwrite the previously saved **/stand/unix.old**.
2. Create the required **Master** and **System** files (these are described in Chapter 3), and put them along with your **Driver.o** device driver module into the **/tmp** directory.
3. You can also create the **Mtune**, **Node**, and other optional DSP component files if needed. However, if possible, you should test your driver first in as simple an environment as possible.
4. Use the **idinstall -a** command to install the new driver.
5. Use the **idbuild** command (with the appropriate options, depending on whether or not your device driver is to be loadable or static) to rebuild the UNIX system kernel.

   If you get errors, correct them and repeat the above step. If the kernel built correctly, a new UNIX system image will have been created. Running **shutdown** will cause the system to enter init state **2**, and the new kernel will be automatically linked to **/stand/unix**. On the next boot, if you specify **/stand/unix** on the **boot:** prompt, the new kernel will execute, and upon entering init state **2**, the new device nodes, **inittab** entries, and so on, will be installed.
6. When the system comes up, test your driver.

Emergency Recovery (New Kernel Will Not Boot)

There is a possibility that the kernel will fail to boot if your driver contains a serious bug. This can be due to a **panic**(D3X) call that you put in your driver or some other system problem. If this happens, you should reset your system and boot your original kernel that you hopefully saved as recommended above. To do this, reset your machine, and when you see the "**Booting UNIX System . . .**" message, quickly strike the keyboard space bar to interrupt the default boot. When the boot prompt appears, type the name of a backup copy of the kernel (for

example, **/stand/unix.bak** or whatever you named your old kernel). If you did not save a copy of your kernel or some other disaster occurred, you can recover the system using the emergency kernel recovery procedure listed in Chapter 3, "Driver Installation and Tuning".

Common Driver Problems

Following is a discussion of some common drivers bugs, with possible symptoms. These should be used only as suggestions. Each driver is unique and will have unique bugs.

Coding Problems

Simple coding problems usually show up when you try to compile the driver. In general, these are similar to coding problems for any C program, such as failure to `#include` necessary header files, define all data structures, or properly delineate comment lines. Specific coding errors unique to driver code include the following:

- `ifdef`-related problems, such as not providing for certain combinations
- inadequate handling of error legs

C Optimizer Bugs

The optimizer (`-O` option to `cc`(1)) on all CPLU 4 releases can be used on drivers without causing problems. However, some old versions of the C optimizer cause problems when used on driver code. For instance, assume a device register is being set to `0` inside a loop, the register is not accessed anywhere else in the loop, and that the register must be set to `0` for every iteration of the loop. The optimizer pulls the statement that initializes the variable to just before the loop, which results in a bug in the driver. Disassembly, using either the `dis`(1) command or the `crash dis` command, can identify such problems.

Installation Problems

Installation problems refer to problems that prevent a system boot with your device configured. If the system won't boot, first try to boot it without the driver to verify that the driver is the problem. Some driver problems that prevent a system boot include:

- Missing information in the `Master` file. Specifically, external variables that are not defined in the `Master` file will not be detected when the driver is compiled, but will cause the following `lboot` error message:

```
symbol undefined - set to zero
```

and will probably cause a kernel MMU panic when the variable is referenced.

- Errors in the **init** or **start** routine. You can check that the initialization routine is being entered by inserting an unconditional **cmn_err** statement at the beginning of the routine.
- Allocating an array in the **Master** file, then not declaring it as a global data structure for the driver or initializing it in an **init** or **start** routine. This will not prevent you from booting the system the first time, but may preclude a reboot from a **/stand/unix** file.

Data Structure Problems

A driver can corrupt the kernel data structures. If the driver is setting or clearing the wrong bits in a device register, a **write** operation may put bad data on the device and a **read** operation may put bad data anywhere in the kernel. Such errors may affect other drivers on the system. Finding this bug involves painstaking walk-throughs of the code. Look for a place where perhaps a pointer is freed (or never set) before the driver tries to access it, or places where the code forgets to check a flag before accessing a certain structure.

Value of Initialized Global Variables

The driver should not depend on initialized global variables having the value assigned them in the driver source file. When the system is booted in absolute mode (from a **/stand/unix** file), driver global variables that are not explicitly initialized will be in **.bss** and will be **0**. Global variables with initializers will be in **.data** and will have whatever value they had at the time the **/stand/unix** file was created.

Timing Errors

Timing errors occur when the driver code executes too quickly or too slowly for the device being driven. For instance, the driver might read a status register on a device too soon after sending the device a command. The device may not have had time to update the status register, so the status register is perceived by the driver to be all `0` bits when, in fact, the device may just be slow in posting the correct status register setting.

When testing the driver, it is useful to verify that a simple, single interrupt is being handled properly. After this is confirmed, you should check that the interrupt handler can handle a number of interrupts that happen at almost the same time.

Corrupted Interrupt Stack

If a driver's interrupt handler runs at an execution level lower than the corresponding IPL for the device, the processing of one interrupt may be interrupted by a second interrupt from the same device. This will seriously corrupt the interrupt stack, which may cause the system to panic with a stack fault or kernel MMU fault. Sometimes, however, it will only cause random operational irregularities, which can make this a difficult problem to detect. You can identify this problem by looking at the interrupt stack in the system dump. If it is corrupted, check the execution level of the driver's interrupt handling routine.

Accessing Critical Data

Check the driver code for data structures that are accessible to both the base and interrupt levels of the driver. Ensure that any section of the base-level code that accesses such structures cannot be interrupted during that access by using the **`spl`***n*(D3X) function.

Overuse of Local Driver Storage

If the driver routines use large amounts of local storage, they may exceed the bounds of the kernel stack or the interrupt stack, which in turn will panic the system.

Incorrect DMA Address Mapping

Failure to set up address mapping for DMA transfers correctly is another common mistake. On a **read** operation, a bad address map may cause data to be placed in the wrong location in the main store, overwriting whatever is there including, for example, a portion of the operating system text.

To check for this, write a simple user program that writes data to all possible memory locations (including shared memory, stack, and text), then reads it back and compares the input and output. As soon as any one of these operations fails, you should reboot the system immediately to ensure that kernel memory is sane.

Testing the Hardware

In addition to testing and debugging the driver, you must also test the hardware device itself. While the area of developing, testing, and debugging the hardware is beyond the scope of this book, the following guidelines are suggested:

- Very early in the development process, you should get the equipment and do some basic tests on its integrity, such as ensuring that it can be powered up without problems and access registers on the peripherals. If the device does not pass these tests, it can be returned to the vendor for further development while you write the driver.
- Write a stand-alone board exerciser that runs at the firmware level (not under the UNIX operating system) to detect hardware bugs. This is an interactive program that is used to exercise a board under controlled conditions. The device should pass these tests before you attempt to test it with your driver.
- Test the diagnostics that are hard-coded on the board by corrupting the hardware and booting the system. Check that the diagnostics detect the corruption and that the messages are sufficient to indicate the maintenance that is required. Power-up diagnostics should verify sanity at a gross level. Demand-phase diagnostics should be used for more extensive checks on the board, such as identifying marginal or intermittent errors.

To ensure that the kernel-device interface is functioning properly, write a simplified driver that contains dummy routine calls for the **init**(D2X), **start**(D2X), **open**(D2X), **close**(D2X), **read**(D2X), and **write**(D2X) routines. For example:

```
qq_open()

    {
    cmn_err(CE_CONT,"Open routine entered\n");
    }
```

This simplified driver should contain an **ioctl**(D2X) routine that gives user program control to each control bit in the control status register (CSR). This lets you test each hardware function and ensure that the hardware is performing in the proper operational sequence. The exact layout of the CSR is specified in the **/usr/include/sys/cc.h** file.

Using crash to Debug a Driver

The **crash**(1M) utility allows you to analyze how your driver interacts with the core image of the operating system. It is most frequently used in postmortem analysis of a system panic, but can also be run on an active system. The output from **crash** can help you identify such driver errors as corrupted data structures and pointers to the wrong address. Its shortcoming as a debugging tool is that it is difficult to freeze the core image at exactly the point where the error occurred; even if the error causes a system panic, the core image may be from beyond the point of actual error. This is especially true when debugging an intelligent board, because an autonomous intelligent controller continues processing even though you have halted kernel-level processing on the main memory. Moreover, for intelligent boards, the **crash** dump cannot get at the onboard data structures.

Using the **crash** command requires a thorough knowledge of assembler, of reading core dumps, and of systems programming concepts. The need to know assembler cannot be overemphasized. The **crash** output is displayed in assembler mnemonics and as strings of hex numbers that must be translated into address locations, stack frames, and memory offsets.

Saving the Core Image of Memory

To run **crash** as a postmortem analysis on a panicked system, you must save the core image of memory before rebooting the system and have a copy of the bootable kernel image (**/stand/unix** file) that was running.

On computers using UNIX System V, the system automatically saves the dump image when it detects an improper shutdown. The partition used by the system to store the dump image is also shared by the swap facility used by the system pager when the computer is in multiuser state. Therefore, do not progress to multiuser state until after you have saved the memory core image to tape or floppy disk. However, saving the core image is only useful if you want to use **crash** to examine it. Saving the dumped memory image is not required and no system software will be damaged if you continue on to multiuser state.

NOTE

If you are familiar with how memory is added to your computer, you can remove excess RAM cards before the system crash to reduce the amount of memory to be copied to disk.

When you try to reboot the system, the following message is displayed automatically.

```
There may be a system dump memory image on the swap device.
Do you want to save it? (y/n)>
```

Answer **y** to save the dump file. When given a selection list of what media to use for the dump, enter the appropriate value for the media you intend to use.

Once booted, you can use the command **ldsysdump** to load the dump file from the tape or disks onto a file system.

Initializing crash on the Memory Dump

To run **crash** on the core image of memory at the time the system panicked, you must have saved the core image before rebooting and the file containing the kernel bootable image (**/stand/unix** file by default) that was running at the time of the crash.

If the bootable kernel image is named something other than **/stand/unix** (either because it was named something else at the time of the panic or because you copied it to another name after the panic), use the **-n** option or the second positional parameter to specify that file name. If you want the output of **crash** to be written to a file rather than your terminal (standard output), use the **-w** option with the name of the file. Note that the output of a specific **crash** command can be redirected to a file even if you do not use the **-w** in the **crash** command line.

Using crash Functions

The **crash** session begins by reporting the *dumpfile, namelist,* and *outfile* being used, followed by the **crash** prompt (>). Requests in the **crash** session have the following standard format

command [*argument. . .*]

where *command* is one of the supported commands of **crash** and *argument* includes any qualifying data relevant to the requested command. Use the **q** command to end the **crash** session.

See the **crash**(1M) manual page for a list of supported commands. Note that, while most **crash** commands are common to all computers, each system also has unique commands that relate to specific devices supported on that machine.

Following is a list of **crash** commands often useful when debugging a driver.

dis Disassemble from a starting address. Use this information to trace code flow. However, you will have to mentally convert the resulting assembler code to C programming language statements.

od List memory. Use this command when you suspect that the stack is corrupted, or to list the contents of memory at a certain address. If you are listing the contents of the stack, you will have to manually find the boundaries of each stack entry, called *stack frames*. To get the starting address of the stack, list the registers with the **panic** command.

proc List the process table. Use this information to obtain the process slot number of the process that panicked the system.

stack Dump the stack. Use this information to determine the size of the stack frame. If **stack** returns information that you suspect is corrupted, use **proc** to get a list of process table slots and then use **stack** on each individual slot entry.

stat List system statistics. Use this information to display the reason a panic occurred. The **panic** command gives the same information as **stat**, plus registers, stack, and trace data.

trace Print kernel stack trace. Use this information to determine which commands were executed in the stack or in an individual process table slot entry.

Using crash Commands

When a panic occurs, capture the core memory image and produce a file that you can use with **crash**. When **crash** executes, a ">" command line prompt is displayed. The following sequence of commands are frequently used to analyze the problem.

1. **stat** — list reason for the crash
2. **proc** — list the process table to see which process initiated the panic

3. **stack** or **trace** — list the last processes on the stack
4. **dis** — trace the execution of a set of instructions

The crash Command in STREAMS

The following **crash** functions are related to STREAMS.

linkblk Print the **linkblk** table.

pty Print pseudo ttys now configured. The **-l** option gives information on the line discipline module **ldterm**, the **-h** option provides information on the pseudo-tty emulation module **ptem**, and the **-s** option gives information on the packet module **pckt**.

qrun Print a list of scheduled queues.

queue Print STREAMS queues.

stream Print the **stdata** table.

strstat Print STREAMS statistics.

tty Print the tty table. The **-l** option prints out details about the line discipline module.

The **crash** functions **linkblk**, **queue**, and **stream** take an optional address that is the address of the data structure. The **strstat** command gives information about STREAMS event cells and **linkblk**s in addition to message blocks, data blocks, queues, and Streams. On the output report, the **CONFIG** column represents the number of structures currently configured. It may change because resources are allocated as needed.

The following example illustrates the debugging of a line printer. Knowledge of the data structures of the driver is needed for debugging. The example starts with the following data structure of the line printer driver.

```
struct lp {
        short lp_flags;
        queue_t *lp_qptr;  /* back pointer to write queue */
};
extern struct lp lp_lp[];
```

The first command, **nm lp_lp**, prints the value and type for the line printer driver data structure. The second command, **rd 40275750 20**, prints 20 values starting from the location 40275750 (note that the function **rd** is an alias of **od**). The third command, **size queue**, gives the size of the **queue** structure. The next two functions again give the 20 values starting at the specified locations in the hexadecimal format. The command **rd -c 4032bf40 32** gives the character representation of the value in the given location. The option **-x** gives a value in the hexadecimal representation and the option **-a** produces the same in the ASCII format.

```
/usr/sbin/crash
dumpfile - /dev/mem, namelist = /stand/unix, outfile = crash.out

> nm lp_lp
lp_lp   40275750   bss

> rd 40275750 20
40275750:  00000000   00000000   00000000   40262f60
40275760:  00000000   00000000   00000000   00000000
40275770:  00000000   00000000   00000000   00000000
40275780:  00000000   00000000   00000000   00000000
40275790:  00000000   00000000   00000000   00000000

> size queue
36

> rd 40262f60 20
40262f60:  4017315c  402624a4  4026257c  00000000
40262f70:  00000000  40275758  0200002e  00000200
40262f80:  02000100  00000000  00000000  00000000
40262f90:  00000000  00000000  00000000  00000000
40262fa0:  00000000  00000000  00000000  00000000

> rd 402624a4 20
402624a4:  40262624  00000000  00000000  4032bf40
402624b4:  4032bf5f  40236884  4026233c  00000000
402624c4:  00000000  40331fd9  40331fd9  00000000
402624d4:  00000000  00000000  00000000  4032bf80
402623e4:  4032bf80  40236894  40262564  00000000

> rd -c 4032bf40 32
4032bf40:  l i t t l e    r e d   l i g h t
4042bf50:    o n   t h e   h i g h w a y \n

> rd -x 40262624 20
40252624:  40262594  402624a4  00000000  4032bd40
40262634:  4032bd5f  40236804  00000000  00000000
40262644:  00000000  4030c800  4030c800  402319e4
40262654:  00000000  00000000  00000000  4032be40
40262664:  4032be40  40236844  4026239c  00000000

> rd -a 4032bd40 31
little red light on the highway
```

Debugging STREAMS Drivers

This section provides some tools to assist in debugging STREAMS-based applications. For detailed information about STREAMS programming and debugging, however, refer to the guide, *STREAMS Modules and Drivers*.

STREAMS modules and drivers can record trace messages using the **strlog**(D3X) function. Calls to this function are converted into STREAMS messages and relayed by **log**(7), a software driver, to the **strace**(1M) process. The **log** driver is also used to send error messages to the **strerr**(D3X) process.

Module and driver writers should limit the number of messages sent to either the error or trace loggers. If a large number of messages are sent some could be lost, because some parts of this facility do not include flow control.

Also, messages may not be delivered to **strace** in the same order in which they were sent. However, every message includes a *sequence number* field provided to make it possible to determine the correct message order where necessary.

STREAMS Debugging

The kernel routine **cmn_err** allows printing of formatted strings on a system console. It displays a specified message on the console and/or stores it in the **putbuf** that is a circular array in the kernel and contains output from **cmn_err**. Its format is:

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

void cmn_err (int level, char *fmt, int ARGS)
```

where *level* can take the following values:

CE_CONT Use as a simple **printf** to continue another message or to display an informative message not associated with an error.

CE_NOTE Report system events. It is used to display a message preceded with **NOTICE:**. This message is used to report system events that do not necessarily require user action, but may interest the system administrator. For example, a sector on a disk needing to be accessed repeatedly before it can be accessed correctly might be such an event.

CE_WARN Report system events that require user action. This is used to display a message preceded with **WARNING:**. This message is used to report system events that require immediate attention, such as those where if an action is not taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.

CE_PANIC Panic the system. This is used to display a message preceded with **PANIC:**. Drivers should specify this level only under the most severe conditions. A valid use of this level is when the system cannot continue to function. If the error is recoverable, not essential to continued system operation, do not panic the system. This level halts all processing.

fmt and *ARGS* are passed to the kernel routine **printf**, which runs at **splhi** and should be used sparingly.

Valid *fmt* specifications are **%s** (string), **%u** (unsigned decimal), **%d** (decimal), **%o** (octal), and **%x** (hexadecimal). **cmn_err** does not accept length specifications in conversion specifications. For example, **%3d** is ignored. If the first character of *fmt* begins with "**!**" (an exclamation point), output is directed to **putbuf**. **putbuf** can be accessed with the **crash**(1M) command. If the destination character begins with "^" (a caret), output goes to the console. If no destination character is specified, the message is directed to both the **putbuf** array and the console. **cmn_err** appends each *fmt* with "**\n**," except for the **CE_CONT** level, even when a message is sent to the **putbuf** array.

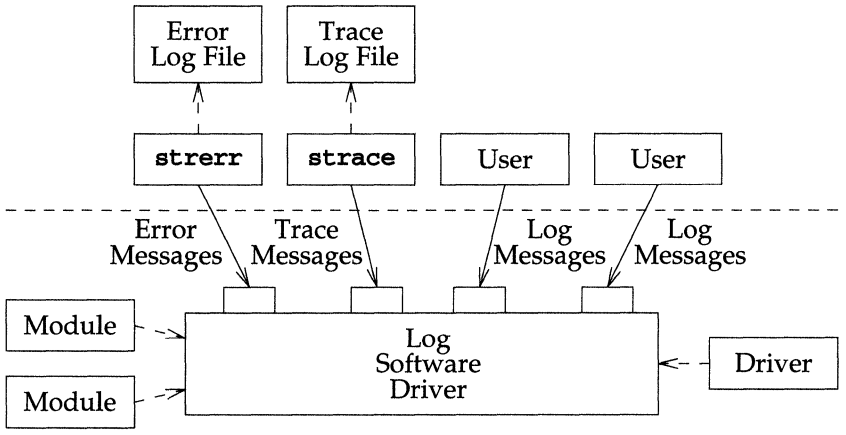
ARGS specifies a set arguments passed when the message is displayed.

STREAMS Error and Trace Logging

STREAMS error and trace loggers are provided for debugging and for administering STREAMS modules and drivers. This facility consists of **log**(7), **strace**(1M), **strclean**(1M), **strerr**(1M), and the **strlog** function.

Any module or driver in any Stream can call the STREAMS logging function **strlog**, described in **log**(7). When called, **strlog** sends formatted text to the error logger **strerr**(1M), the trace logger **strace**(1M), or the console logger.

Figure 4-1: Error and Trace Logging



strerr is intended to operate as a daemon process initiated at system startup. A call to **strlog** requesting an error to be logged causes an **M_PROTO** message to be sent to **strerr**, which formats the contents and places them in a daily file. The utility **strclean**(1M) is provided to purge daily log files that have not been modified for three days.

A call to **strlog** requesting trace information to be logged causes a similar **M_PROTO** message to be sent to **strace**(1M), which places it in a user designated file. **strace** is intended to be initiated by a user. The user can designate the modules/drivers and severity level of the messages to be accepted for logging by **strace**.

A user process can submit its own **M_PROTO** messages to the log driver for inclusion in the logger of its choice through **putmsg**(2). The messages must be in the same format required by the logging processes and will be switched to the logger(s) requested in the message.

The output to the log files is formatted, ASCII text. The files can be processed by standard system commands such as **grep**(1) or **ed**(1), or by developer-provided routines.

Driver Debugging Techniques

Kernel Print Statements

There are, of course, limitations in debugging and testing device drivers. If the kernel debugger, **kdb**, is not available, print statements inside the driver are the primary method used. Because the print statements are written by the kernel, there is no way to redirect the output to a file or to a remote terminal. Using print statements also modifies the timing of driver code execution, which may change the behavior of problems you are investigating. Print statements in the driver can be made more efficient by using an **ioctl** to set one or more levels of debugging output. This way you can write a simple user program to turn the print output on or off as needed.

System Panics

If you expect that the driver could enter a state that is invalid, the driver can halt the system by using the **cmn_err** function with a panic flag set. For example, if the driver expects one of three specific cases in a **switch** statement, the driver can add a fourth default case that calls the **cmn_err** function. The system will dump an image of memory for later analysis. If the error is recoverable, the driver should not panic the system. An example of panicking using **cmn_err** is

```
cmn_err(CE_PANIC,"Your system has panicked, DEV_NAME error!");
```

Taking a System Dump

In the event a **panic** occurs, there may be some value in examining the dump produced by the system. Because UNIX System V uses the same physical hard disk partition for both "swap" and "dump," it is important that you do not reboot to the multiuser state before examining the dump. If the system reaches multiuser state, the dump may be overwritten by system paging.

To examine the dump, the dump image must be saved. If the **root** partition does not have enough space to save the crash dump, the following message will appear.

```
Need nnnnK to save crash dump.
Root has only xxxxK free.
F - write to floppy disk
T - write to tape
S - spawn a shell
X - skip it
```

You may then proceed in whatever manner you prefer.

NOTE

We recommend that you write your crash dump to tape.

When the system reboots and detects a dump image, it will copy the dump image from the **swap/dump** area to the file **crash.***MMDD* in the **/crash** directory; where *MM* is the month, and *DD* is the day. If a crash file already exists in the **/crash** directory, another crash file is created with a **.1, .2, .3**, and so forth appended to the file name. A corresponding symbol file, **sym.***MMDD* is also saved in the **/crash** directory.

Before the dump image is saved, the following message appears on the console.

Saving *nnnn***K crash dump in crash.***MMDD*

where *nnnn* is the size of the dump in KB. After the dump image is saved, the console displays the following message, **Done**, and the system continues its start-up procedure.

You can use the **crash** command to examine the dump as follows.

crash -d *dump-file* **-n** *symbol-file*

or you can use the **kcrash** command to examine the dump as follows.

kcrash *dump-file symbol-file*

Consult the **crash**(1M) and **kcrash**(1M) manual pages in the UNIX System V *Command Reference* for information on how to use **crash** and **kcrash** to examine the UNIX operating system kernel and user process status at the time of the **panic**.

Note that the procedures to examine a memory dump only apply to systems that have completed the dump sequence, usually in response to a **panic**. The prompt that you may see after an improper shutdown only indicates that the system was not properly brought down and a dump may exist. If the system is inadvertently powered down or reset, or if your device driver causes the kernel to hang or go

berserk without ever executing a **panic**, no dump will have been taken. Remember, the system will only do a dump when you have properly detected an error and executed the **panic** function inside your driver or when your driver has caused a system error detected by the kernel or some other driver causing it to **panic**.

At this point, it might be well to repeat the advice stated in the introduction:

Writing a device driver carries a heavy responsibility. As part of the UNIX system kernel, it is assumed to always take the correct action. Few limits are placed on the driver by the other parts of the kernel, and the driver must be written to never compromise the system's stability.

Kernel Debugger

An extremely useful tool for debugging device drivers is the kernel debugger (-**kdb**). Refer to the **kdb**(1) manual page in the *Command Reference* for more detail and a complete list of commands for the **kdb** utility.

kdb can set breakpoints, display kernel stack traces and various kernel structures, and modify the contents of memory, I/O, and registers. The debugger supports basic arithmetic operations, conditional execution, variables, and macros. **kdb** does conversions from a kernel symbol name to its virtual address, from a virtual address to the value at that address, and from a virtual address to the name of the nearest kernel symbol. You have a choice of different numeric bases, address spaces, and operand sizes.

You can invoke the debugger by using the **kdb** command or the **sysi86(SI86TODEMON)** system call, or by pressing **CTRL-ALT-d** (from the console only) on an AT-bus system, or by typing the interrupt character (from the console only) on a Multibus system. In addition, **kdb** is entered automatically under various conditions, such as panics and breakpoint traps. Any time the **kdb>>** prompt appears, you are in the debugger. I/O is performed through the console or a serial terminal.

To exit the debugger, press **CTRL-d** or **q**.

When you exit and re-enter the debugger, its state is preserved, including the contents of the value stack.

kdb is an extremely powerful tool, and should be used carefully to avoid accidental corruption of kernel data structures, which could lead to a system crash. **kdb** has few provisions for preventing programmer error.

The kernel debugger is not meant for debugging user programs. Use an appropriate user-level debugger, such as **sdb**(1), for that purpose.

kdb must exist in your kernel before you can use it (just like any device driver).

kdb prints and accepts address inputs symbolically, using kernel procedure and variable names instead of hexadecimal numbers, but you must load the debugger with the kernel's symbols after the debugger itself has been installed into the kernel. You can do this by using the **unixsyms** command, which loads the symbols into the kernel executable file after building it and before booting it. Normally, this will be done automatically for you by **idbuild**(1M).

NOTE

The symbols must be loaded before the system panics (or you enter the kernel debugger for some other reason) for them to be useful. You cannot load the kernel symbols while in the debugger.

Entering kdb from a Driver

If you are debugging a device driver or another part of the kernel, you can directly invoke the kernel debugger by including this code in your driver.

```
#include <sys/xdebug.h>
(*cdebugger) (DR_OTHER, NO_FRAME);
```

DR_OTHER tells **kdb** that the reason for entering is "other." See **sys/xdebug.h** for a list of other reason codes.

Note that this mechanism cannot be used for debugging early kernel startup code or driver **init** routines, since the debugger cannot be used until its **init** routine (**kdb_init**) has been called.

GL Glossary

Glossary

The following is a list of terms used throughout the *Device Driver Programming* document set:

alignment	The position in memory of a unit of data, such as a word or half-word, on an integral boundary. A data unit is properly aligned if its address is evenly divisible by the data unit's size in bytes. For example, a word is correctly aligned if its address is divisible by four. A half-word is aligned if its address is divisible by two.
ARP	Address Resolution Protocol
asm macro	The macro that defines system functions used to improve driver execution speed. They are assembler language code sections (instead of C code).
asynchronous	An event occurring in an unpredictable fashion. A signal is an example of an asynchronous event. A signal can occur when something in the system fails, but it is not known when the failure will occur.
automatic calling unit (ACU)	A device that permits processors to dial calls automatically over the communications network.
base level	The code that synchronously interacts with a user program. The driver's initialization and switch table entry point routines constitute the base level. Compare **`interrupt level`**.
block and character interface	A collection of driver routines, kernel functions, and data structures that provide a standard interface for writing block and character drivers.
block data transfer	The method of transferring data in units (blocks) between a block device such as a magnetic tape drive or disk drive and a user program.

block device switch table	The table constructed during automatic configuration that contains the address of each block driver entry point routine [for example, **open**(D2DK), **close**(D2DK), **strategy**(D2DK)]. This table is called **bdevsw** and its structure is defined in **conf.h**.
block device	A device, such as a magnetic tape drive or disk drive, that conveys data in blocks through the buffer management code. Compare **character device**.
block driver	A device driver, such as for a magnetic tape device or disk drive, that conveys data in blocks through the buffer management code (for example, the **buf** structure). One driver is written for each major number employed by block devices.
block I/O	A data transfer method used by drivers for block access devices. Block I/O uses the system buffer cache as an intermediate data storage area between user memory and the device.
block	The basic unit of data for I/O access. A block is measured in bytes. The size of a block differs between computers, file system sizes, or devices.
boot device	The device that stores the self-configuration and system initialization code and necessary file systems to start the operating system.
bootable object file	A file that is created and used to build a new version of the operating system.
bootstrap	The process of bringing up the operating system by its own action. The first few instructions load the rest of the operating system into the computer.
boot	The process of starting the operating system. The boot process consists of self-configuration and system initialization.

buffer	A staging area for input-output (I/O) processes where arbitrary-length transactions are collected into convenient units for system operations. A buffer consists of two parts: a memory array that contains data from the disk and a buffer header that identifies the buffer.
cache	A section of computer memory where the most recently used buffers, i-nodes, pages, and so on are stored for quick access.
called DLS user	The DLS user in connection mode that processes requests for connections from other DLS users.
calling DLS user	The DLS user in connection mode that initiates the establishment of a data link connection.
canonical processing	Terminal character processing in which the erase character, delete, and other commands are applied to the data received from a terminal before the data is sent to a receiving program. Other terms used in this context are canonical queue, which is a buffer used to retain information while it is being canonically processed, and canonical mode, which is the state where canonical processing takes place. Compare **`raw mode`**.
character device	A device, such as a terminal or printer, that conveys data character by character. Compare **`block device`**.
character driver	The driver that conveys data character by character between the device and the user program. Character drivers are usually written for use with terminals, printers, and network devices, although block devices, such as tapes and disks, also support character access.
character I/O	The process of reading and writing to/from a terminal.
CLNS	Connectionless Network Service, the datagram version of the OSI network layer

clone driver	A software driver used by STREAMS drivers to select an unused minor device number, so that the user process does not need to specify it.
communication endpoint	The local communication channel between a DLS user and DLS provider.
connection establishment	The phase in connection mode that enables two DLS users to create a data link connection between them.
connection management stream	A special stream that will receive all incoming connect indications destined for Data Link Service Access Point (DLSAP) addresses that are not bound to any other streams associated with a particular Physical Point of Attachment (PPA).
connection mode	A circuit-oriented mode of transfer in which data is passed from one user to another over an established connection in a sequenced manner.
connection release	The phase in connection mode that terminates a previously established data link connection.
connectionless mode	A mode of transfer in which data is passed from one user to another in self-contained units with no logical relationship required among the units.
control and status register (CSR)	Memory locations providing communication between the device and the driver. The driver sends control information to the CSR, and the device reports its current status to it.
controller	The circuit board that connects a device, such as a terminal or disk drive, to a computer. A controller converts software commands from a driver into hardware commands that the device understands. For example, on a disk drive, the controller accepts a request to read a file and converts the request into hardware commands to have the reading apparatus move to the precise location and send the information until a delimiter is reached.

critical code	A section of code is critical if execution of arbitrary interrupt handlers could result in consistency problems. The kernel raises the processor execution level to prevent interrupts during a critical code section.
CSMA/CD	Carrier Sense Multiple Access/Collision Detection
cyclic redundancy check (CRC)	A way to check the transfer of information over a channel. When the message is received, the computer calculates the remainder and checks it against the transmitted remainder.
data structure	The memory storage area that holds data types, such as integers and strings, or an array of integers. The data structures associated with drivers are used as buffers for holding data being moved between user data space and the device, as flags for indicating error device status, as pointers to link buffers together, and so on.
data terminal ready (DTR)	The signal that a terminal device sends to a host computer to indicate that a terminal is ready to receive data.
data transfer	The phase in connection and connectionless modes that supports the transfer of data between two DLS users.
DDI/DKI	Device Driver Interface/Device Kernel Interface
demand paging	A memory management system that allows unused portions of a program to be stored temporarily on disk to make room for urgently needed information in main memory. With demand paging, the virtual size of a process can exceed the amount of physical memory available in a system.
device number	The value used by the operating system to name a device. The device number contains the major number and the minor number.

dev_t	The C programming language data type declaration that is used to store the driver major and the minor device numbers.
diagnostic	A software routine for testing, identifying, and isolating a hardware error. A message is generated to notify the tester of the results.
DLIDU	Data Link Interface Data Unit. A grouping of DLS user data that is passed between a DLS user and the DLS provider across the data link interface. In connection mode, a DLSDU may consist of multiple DLIDUs.
DLM	Dynamically Loadable Modules
DLPI	Data Link Provider Interface
DLS provider	The data link layer protocol that provides the services of the Data Link Provider Interface.
DLS user	The user-level application or user-level or kernel-level protocol that accesses the services of the data link layer.
DLS	Data Link Service
DLSAP address	An identifier used to differentiate and locate specific DLS user access points to a DLS provider.
DLSAP	A point at which a DLS user attaches itself to a DLS provider to access data link services.
DLSDU	Data Link Service Data Unit. A grouping of DLS user data whose boundaries are preserved from one end of a data link connection to the other.
downstream	The direction of STREAMS messages flowing through a write queue from the user process to the driver.
driver entry points	Driver routines that provide an interface between the kernel and the device driver.
driver routines	See **routines**.

driver	The set of routines and data structures installed in the kernel that provide an interface between the kernel and a device.
DSAP	Destination Service Access Point
EDLIDU	Expedited Data Link Interface Data Unit
error correction code (ECC)	A generic term applied to coding schemes that allow for the correction of errors in one or more bits of a word of data.
expedited data transfer	A DLPI service that transfers data subject to separate flow control than that applying to normal data transfer. The service is intended to deliver the data ahead of any DLSDUs that may be in transit.
FDDI	Fiber Distributed Data Interface
function	A kernel utility used in a driver. The term function is used interchangeably with the term kernel function. The use of functions in a driver is analogous to the use of system calls and library routines in a user-level program.
initialization entry points	Driver initialization routines that are executed during system initialization [for example, **`init`**(D2D), **`start`**(D2DK)].
interface	The set of data structures and functions supported by the UNIX kernel to be used by device drivers.
interprocess communication (IPC)	A set of software-supported facilities that enable independent processes, running at the same time, to share information through messages, semaphores, or shared memory.
interrupt level	Driver interrupt routines that are started when an interrupt is received from a hardware device. The system accesses the interrupt vector table, determines the major number of the device, and passes control to the appropriate interrupt routine.

interrupt priority level (IPL)	The interrupt priority level at which the device requests that the CPU call an interrupt process. This priority can be overridden in the driver's interrupt routine for critical sections of code with the **spl***n*(D3D) function.
interrupt vector	Interrupts from a device are sent to the device's interrupt vector, activating the interrupt entry point for the device.
IP	Internet Protocol
ISO	International Organization for Standardization
kernel buffer cache	A linked list of buffers used to minimize the number of times a block-type device must be accessed.
LLC	Logical Link Control, a sub-layer of the data link layer for media independent data link functions.
loadable module	A kernel module (such as a device driver) that can be added to a running system without rebooting the system or rebuilding the kernel.
low water mark	The point at which more data is requested from a terminal because the amount of data being processed in the character lists has fallen creating room for more. It also applies to STREAMS queues regarding flow control.
MAC	Media Access Control, a sub-layer of the data link layer for media specific data link functions.
memory management	The memory management scheme of the UNIX operating system imposes certain restrictions on drivers that transfer data between devices.
message block	A STREAMS message is made up of one or more message blocks. A message block is referenced by a pointer to a **mblk_t** structure, which in turn points to the data block (**dblk_t**) structure and the data buffer.

message	All information flowing in a stream, including transferred data, control information, queue flushing, errors and signals. The information is referenced by a pointer to a **mblk_t** structure.
MIB	Management Information Base
modem	A contraction of modulator-demodulator. A modulator converts digital signals from the computer into tones that can be transmitted across phone lines. A demodulator converts the tones received from the phone lines into digital signals so that the computer can process the data.
module	A STREAMS module consists of two related **queue** structures, one each for upstream and downstream messages. One or more modules may be pushed onto a stream between the stream head and the driver, usually to implement and isolate a line discipline or a communication protocol. virtual to physical memory.
panic	The state where an unrecoverable error has occurred. Usually, when a panic occurs, a message is displayed on the console to indicate the cause of the problem.
PDU	Protocol Data Unit
portable device interface (PDI)	A collection of driver routines, kernel functions, and data structures that provide a standard interface for writing block drivers.
PPA identifier	An identifier of a particular physical medium over which communication transpires.
PPA	The point at which a system attaches itself to a physical communications medium.
prefix	A character name that uniquely identifies a driver's routines to the kernel. The prefix name starts each routine in a driver. For example, a RAM disk might be given the **ramd** prefix. If it is a block driver, the routines are

	ramdopen, **ramdclose**, **ramdstrategy**, and **ramdprint**.
priority message	STREAMS messages that must move through the stream quickly are classified as priority messages. They are placed at the head of the queue for processing by the **srv**(D2DK) routine.
quality of service (QOS)	Characteristics of transmission quality between two DLS users.
queue	A data structure, the central node of a collection of structures and routines, which makes up half of a STREAMS module or driver. Each module or driver is made up of one queue each for upstream and downstream messages. Location: **stream.h**.
raw I/O	Movement of data directly between user address spaces and the device. Raw I/O is used primarily for administrative functions where the speed of a specific operation is more important than overall system performance.
raw mode	The method of transmitting data from a terminal to a user without processing. This mode is defined in the line discipline modules.
read queue	The half of a STREAMS module or driver that passes messages upstream.
routines	A set of instructions that perform a specific task for a program. Driver code consists of entry-point routines and subordinate routines. Subordinate routines are called by driver entry-point routines. The entry-point routines are accessed through system tables.
SAP	Service Access Point, conceptually the "point" at which a layer in the OSI model make its services available to the layer above it.
SCSI driver interface (SDI)	A collection of machine-independent input/output controls, functions, and data structures, that provide a standard interface for writing Small Computer System Interface (SCSI) drivers.

SDU	Service Data Unit
semantic processing	Semantic processing entails input validation of the characters received from a character device.
small computer system interface (SCSI)	The American National Standards Institute (ANSI) approved interface for supporting specific peripheral devices.
SNMP	Simple Network Management Protocol
Source Code Control System (SCCS)	A utility for tracking, maintaining, and controlling access to source code files.
special device file	The file that identifies the device's access type (block or character), the external major and minor numbers of the device, the device name used by user-level programs, and security control (owner, group, and access permissions) for the device.
SSAP	Source Service Access Point
stream end	The stream end is the component of a stream farthest from the user process, providing the interface to the device. It contains pointers to driver (rather than module) routines.
stream head	Every stream has a stream head, which is inserted by the STREAMS subsystem. It is the component of a stream closest to the user process. The stream head processes STREAMS-related system calls and performs the transfer of data between user and kernel space.
STREAMS	A kernel subsystem used to build a stream, which is a modular, full-duplex data path between a device and a user process.
stream	A linked list of kernel data structures providing a full-duplex data path between a user process and a device or pseudo-device.

switch table entry points	Driver routines that are activated through **bdevsw** or **cdevsw** tables.
switch table	The operating system that has two switch tables, **cdevsw** and **bdevsw**. These tables hold the entry point routines for character and block drivers and are activated by I/O system calls.
synchronous data link interface (SDLI)	A UN-type circuit board that works subordinately to the input/output accelerator (IOA). The SDLI provides up to eight ports for full-duplex synchronous data communication.
system initialization	The routines from the driver code and the information from the master file that initialize the system (including device drivers).
TCP	Transmission Control Protocol, a connection oriented transport in the Internet suite
upstream	The direction of STREAMS messages flowing through a read queue from the driver to the user process.
user space	The part of the operating system where programs that do not have direct access to the kernel structures and services execute. The UNIX operating system is divided into two major areas: the user program and the kernel. Drivers execute in the kernel, and the user programs that interact with drivers generally execute in the user program area. This space is also referred to as user data area.
volume table of contents (VTOC)	Lists the beginning and ending points of the disk partitions by the system administrator for a given disk.
write queue	The half of a STREAMS module or driver that passes messages downstream.

IN Index

Index

A

B

C

D

E

L

M

N

O

P

Q

R

S

T

U

V

W

UNIX® SVR4.2 PUBLISHED BOOKS

User's Series

Guide to the UNIX® Desktop
User's Guide

Administration Series

Basic System Administration
Advanced System Administration
Network Administration
Audit Trail Administration
PC Interface™ Administration

Programming Series

UNIX® Software Development Tools
Programming in Standard C
Programming with UNIX® System Calls
Character User Interface Programming
Graphical User Interface Programming
Network Programming Interfaces
Portable Device Interface (PDI)
Device Driver Programming
STREAMS Modules and Drivers

Reference Series

Command Reference (a-l)
Command Reference (m-z)
Operating System API Reference
Windowing System API Reference
System Files and Devices Reference
Device Driver Reference

New for UNIX® System V Release 4.2, ***Device Driver Programming*** contains the latest information for writing, installing, and testing UNIX System V device drivers. This guide provides an in-depth explanation of new SVR4.2 features such as dynamically loadable kernel modules, the new device driver installation tools, and the new system configuration file formats. Topics include:

- Basic concepts of writing a UNIX System V device driver
- Using the block and character device driver entry points and kernel utility functions
- Learning how to make your device drivers loadable, so you can add your drivers to a running system without rebuilding the kernel, and without having to bring the system down to reboot it (includes sample coding for all loadable module types, including device drivers, STREAMS modules and drivers, and Host Bus Adapter drivers)
- Packaging and installing device drivers (and other types of kernel modules) using the new SVR4.2 driver installation tools
- Understanding the new SVR4.2 system configuration file formats used to link device drivers to the rest of the kernel, and how they differ from those for previous UNIX System V releases
- Testing and debugging device drivers using the `crash` and `kdb` commands

This guide is part of the UNIX System V Programming Series. Other driver-related titles in this series include:

- ***STREAMS Modules and Drivers*** - Intended for network and system programmers, this guide is the definitive source of information for kernel-level STREAMS programming—in both uniprocessor and multiprocessor UNIX System V Release 4 environments. STREAMS is a general, flexible facility for development of input/output services in UNIX System V.
- ***Portable Device Interface*** - New for UNIX System V Release 4.2, this guide explains the PDI, a new programming interface for developing portable block-oriented device drivers. The guide presents in-depth information on using the PDI to organize, simplify, and standardize the way Host Bus Adapter, SCSI and non-SCSI target drivers operate in UNIX System V.

UNIX PRESS
A Prentice Hall Title

ISBN 0-13-042623-7