# Macintosh™ Building A/UX® Device Drivers

## Copyright



© Apple Computer, Inc., 1988
20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010



Simultaneously published in the United States and Canada.

# Preface

# About This Manual

## Inside this manual

This manual explains how to build Apple® A/UX® device drivers for the Apple Macintosh ® II computer. The manual is designed to be both a "how-to" guide and a reference manual for someone writing device drivers. A/UX is Apple's version of the UNIX® operating system.

To use this manual effectively, you should have a working knowledge of the C programming language and written device drivers in the past. You need some knowledge of the A/UX operating system, including the major parts of A/UX, although detailed knowledge of the kernel is not required. If you need to learn more about the A/UX operating system, see the bibliography in the back of this manual. You also need to know how to use system calls in a C program.

An overview of what this manual covers is listed below:

- Chapters 1 and 2 provide an overview of A/UX device drivers and the A/UX kernel programming environment. You should read these sections before writing your driver.
- Chapter 3 describes drivers that buffer data through the kernel buffer cache. These drivers are called block device drivers.
- Chapter 4 describes drivers that use their own techniques to transfer data. These drivers are called character device drivers.
- Chapters 5 through 11 describe specific types of device drivers and interfaces. You need to read only those sections that apply to your device and driver.
- Chapter 12 describes the autoconfiguration process. This chapter tells you how to add a new device driver to the kernel.

- Chapter 13 tells you how to use the autoconfiguration process in a driver development environment. This chapter takes you through all the steps necessary to add a device driver to your system by showing a specific example of adding a driver to the kernel.
- Chapter 14 describes the files you need to include on the distribution floppy disk that your customers use to install your driver. The installation procedure that your customers need to follow to install your driver are also given.
- Appendix A describes the driver interface routines.
- Appendix B describes kernel routines your driver can use.
- Appendix C describes the slot library routines that slot device drivers can use.
- Appendix D contains physical, user, and kernel memory maps.
- Appendix E describes vnode kernel modifications.
- Appendix F describes the differences between the System V Release 2.1 and System V Release 3 Streams implementation.
- Appendix G contains a SCSI device driver listing.

## Conventions used in this manual

Words that you must type exactly as shown or that would actually appear on the screen appear in Courier type. Words that you must replace with actual values appear in *italics* (for example, the integer variable *dev* might have an actual value of 2). An ellipsis (...) follows an argument that may be repeated any number of times. **Boldface** type is used for new terms that are defined in the text; often these terms are listed in the glossary for this manual.

Special keys on the keyboard appear in CAPS AND SMALL CAPS (for example, RETURN).

Key combinations that you must press simultaneously are connected with hyphens (for example, CONTROL-S).

A file is enclosed in angled brackets, for example `<sys/buf.h>`, to indicate the parent directory is `/usr/include`.

### Syntax notation

This manual uses the following conventions to represent command and routine syntax. A typical A/UX routine has the following form:

```
type routine (arg, ...)
```

```
type arg;
```

The elements have these meanings:

*type* is the data type of the value returned from the routine (for example, `int`); *type* also specifies the data type of an argument to the routine.

`routine` is the name of the routine.

*arg* is an argument to the routine.

In the text, *cmd(sect)* indicates a cross-reference to an A/UX reference manual. *cmd* is the name of a command, program, system call, or other facility, and *sect* is the section number where the entry can be found. For example, open(2) refers to the open system call, which is documented in section 2 of the *A/UX Programmer's Reference*.

In the text, kernel routines are denoted by the name of the routine in Courier type followed by on open parentheses and a closed parentheses. For example, `biowait()` refers to a kernel routine that you can use in your driver.

# Contents

# Figures and tables

# Chapter 1

# An Overview of A/UX Device Drivers

This chapter provides an overview of A/UX device drivers. Specifically, you'll learn

- what the general functions of the A/UX kernel are
- how the kernel, device driver, and device interact
- what a device driver is
- what the basic structure of an A/UX device driver is
- what hardware is part of the Macintosh II
- what steps to take to begin writing your driver

First, this chapter briefly describes the A/UX kernel and input/output (I/O).

## An overview of the A/UX kernel

The A/UX kernel is an operating system. Like most operating systems, the A/UX kernel performs file management, memory management, process management, and input and output. The kernel contains all the routines necessary to accomplish these functions. For example, when a program runs, the kernel is responsible for allocating enough memory to the process.

Similarly, the kernel is responsible for managing and performing I/O. The kernel routines for doing I/O include both general routines and specific routines. The kernel uses general routines to manage I/O transfers in a deterministic and consistent manner. The specific routines that perform I/O to a particular piece of hardware are called **device drivers**. In addition, the A/UX kernel supplies a number of routines called **managers.** Managers perform a variety of I/O-related functions. Your device driver can call these manager routines to handle many hardware-related I/O tasks.

Figure 1-1 shows a simplified overview of an I/O request. When a user process requests I/O, the appropriate routines within the kernel carry out the request.

**Figure 1-1**

Overview of kernel management routines

## Performing I/O in A/UX

A device driver provides a connection between a user request for I/O and the hardware operation. This connection is actually comprised of several components:

- a user-level program
- the A/UX kernel
- the device driver code
- a device

A user-level program requests an I/O operation by making a system call. System calls perform operations on behalf of the requesting user process. For example, you can use system calls to prepare a device for I/O, to read from or write to a device, or to perform control functions on a device.

The system calls that you can use to perform I/O are:

- `open(2)`
- `close(2)`
- `read(2)`
- `write(2)`
- `ioctl(2)`
- `select(2)`

When a user program makes a system call requesting I/O, the kernel calls the appropriate device driver. The device driver then takes the necessary actions to perform the actual I/O. Figure 1-2 shows the general flow of an I/O request from the user process to the device.

The kernel has a method of mapping a request to a particular device to the associated device driver that performs the I/O. This mapping is established through device files and kernel data structures called device switch tables.

Every device must have a device file associated with it. A device file contains an index into the device switch table. Pointers to driver routines associated with that device are stored at this index.

**Figure 1-2**
The flow of I/O from a user process to a device

Now that you have a general understanding of device files and device switch tables, the following paragraphs explain the I/O process in greater detail. When a user process makes a system call on a device file or file descriptor associated with the device file, the kernel does initial processing of the request. This initial processing includes process management and file management functions. For example, on an `open(2)` call, the kernel first checks that the requesting user has the proper permissions to access the file.

After this initial processing, the kernel uses the index from the device file to index into a device switch table. The kernel calls the corresponding driver routine stored at this index.

The device driver performs the request and returns to the kernel. The kernel then returns to the user process. The return value of the system call indicates the success or failure of the request.

## What is a device driver?

A device driver is a piece of code that handles all I/O operations to or from a device. The kernel calls a device driver when a user process requests I/O by making a system call. The device driver is responsible for carrying out the I/O request.

Figure 1-3 illustrates that you can use many different devices for I/O on the Macintosh II. Each piece of hardware connected to your computer needs supporting code to control it. For example, if you have a video card installed in your computer and a monitor connected to that video card, you need the software to control that monitor and video card. Typically each type of device has a particular device driver associated with it. For example, the floppy disk driver handles all requests to floppy disks.

Apple Computer supplies certain device drivers as part of the A/UX kernel. These drivers include a device driver for SCSI disks, floppy disks, serial ports, the keyboard, the mouse, and the monitor or system console.

Apple also supplies the low-level routines and managers that control the hardware interface to the system. These routines and managers include the code to control transfers over the NuBus™, the Apple Desktop Bus™ (ADB), the Small Computer System Interface (SCSI), and the Serial Communications Controller (SCC). Your driver must use these low-level routines or managers to control transactions on the hardware interface that connects your device to the computer. These hardware interfaces are discussed in more detail in the section "An Overview of the Hardware" in this chapter.

When you add a new device to the system, you must also add a device driver to control the device and to perform I/O to the device. If a device driver to control the device does not exist, then you must write a new device driver in order to perform I/O to the device.

In A/UX, device drivers are part of the kernel. You can add or remove device drivers from the kernel using the `autoconfig(2)` utility.

A device driver contains various routines used to perform I/O on the device. The following section describes the name and purpose of each routine. In addition, these driver routines can call other kernel routines and make use of low-level routines and managers to assist in performing the I/O operation. The following chapters describe these kernel routines and low-level routines and managers.

**Figure 1-3**
Various devices that can be attached to a Macintosh II

## The basic structure of an A/UX device driver

A/UX uses two kinds of device drivers: block and character. Chapter 2 describes the differences between these two types of device drivers in greater detail. This section describes the various routines that make up a device driver. Both types of device drivers can supply a certain set of routines to the kernel. These routines correspond to the system calls used to perform I/O.

### Block Device Drivers

For each system call used to perform I/O using block device drivers, Table 1-1 lists the corresponding driver routine that the kernel invokes and the function of the driver routine.

**Table 1-1**
System calls and the corresponding driver routines for block device drivers

| System call | Driver routines | Purpose |
|---|---|---|
| `open(2)` | *driver*`open` | Open a device |
| `close(2)` | *driver*`close` | Close a device |
| `read(2)` | *driver*`strategy` | Schedule the transfer of data between the buffer cache and a device |
| `write(2)` | *driver*`strategy` | Schedule the transfer of data between the buffer cache and a device |

You must name the driver routines according to the conventions shown in the table, where *driver* is the device prefix used in your driver. For example, if your device prefix is `disk`, then name your *driver*`open` routine `diskopen`.

Block device drivers also provide a *driver*`print` routine. This routine is not related to a system call.

Block device drivers can also provide an optional routine to perform initialization functions. This routine is named *driver*`init`, where *driver* is the device prefix used in your driver.

Block device drivers also can provide an interrupt routine. This routine is named *driver*`int`, where *driver* is the device prefix used in your driver.

## Character Device Drivers

For each system call used to perform I/O using character device drivers, Table 1-2 lists the corresponding driver routine that the kernel invokes and the function of the driver routine.

**Table 1-2**
System calls and the corresponding driver routines for character device drivers

| System call | Driver routines | Purpose |
|---|---|---|
| `open(2)` | *driver*`open` | Open a device |
| `close(2)` | *driver*`close` | Close a device |
| `read(2)` | *driver*`read` | Read from the device |
| `write(2)` | *driver*`write` | Write to the device |
| `ioctl(2)` | *driver*`ioctl` | Perform control operations on the device |
| `select(2)` | *driver*`select` | Check a device for I/O |

You must name the driver routines according to the conventions shown in the table, where *driver* is the device prefix used in your driver. For example, if your device prefix is `mouse`, then name your *driver*`open` routine `mouseopen`.

Character device drivers can also provide an interrupt routine. This routine is named *driver*`int`, where *driver* is the device prefix used in your driver. Character device drivers can provide an optional routine to perform initialization functions. This routine is named *driver*`init`, where *driver* is the device prefix used in your driver.

The following chapters describe each of these routines and how to write these routines for your driver. Appendix A also includes descriptions of these routines, including parameters and return values. The following section discusses the various hardware interfaces on the Macintosh II, and gives examples of the structure of a typical device driver for each hardware interface.

## An overview of the hardware

To understand the complete hardware path to your device, refer to Figure 1-4. This figure shows that the Macintosh II contains more than one bus or hardware interface that can be used for I/O. These hardware interfaces include the NuBus, Small Computer System Interface (SCSI), Versatile Interface Adapters (VIA), Apple Desktop Bus (ADB), Integrated Woz Machine (IWM), Apple Sound Chip (ASC) and Serial Communications Controller (SCC). Each of these is discussed briefly in this section.

### The NuBus

The NuBus is a 32-bit wide address and data bus based on a Texas Instruments specification. Six expansion slots are available for NuBus cards. Examples of cards that can go in NuBus slots are video cards, processor cards, network cards, and other I/O cards. You can connect a wide variety of devices to various NuBus cards.

The A/UX kernel supplies a set of routines called the Slot Library. Routines in the Slot Library can be used to assist in reading information from the slot ROM on your card. For example, if you are writing a slot device driver, you can use the Slot Library to read the resource directory from a slot ROM.

To write a device driver for a NuBus card, you write the high-level code to perform the I/O to the NuBus card, including any card and device specific code. Figure 1-5 illustrates the structure of a device driver for a device connected to a NuBus card. To perform I/O to the device, a device driver must control the I/O from the kernel level, to the NuBus, to the NuBus card, and then to the device.

Your driver can call Slot Library routines to assist in accessing slot ROM. This greatly simplifies the task of writing a device driver for a device on a NuBus card. The Slot Library is described in Chapter 9 and Appendix C.

**Figure 1-4**
Overview of the Macintosh II architecture

**Figure 1-5**
The structure of a typical NuBus slot device driver

## The Small Computer System Interface (SCSI)

The built-in SCSI port is used for high-speed parallel communications. The SCSI chip can communicate with up to seven SCSI devices, such as hard disks, streaming tapes, and high-speed printers. The SCSI Manager supports the NCR 5380 SCSI chip in software. The SCSI Manager takes care of the low-level hardware aspects of controlling the SCSI bus.

Figure 1-6 illustrates the structure of a device driver that controls a disk drive connected to the SCSI bus. To perform I/O to a device connected to the SCSI bus, a driver must control the I/O from the kernel level, to the SCSI bus, and to the SCSI device. A SCSI device driver contains the code to process the data according to the requirements of the device, and calls routines in the SCSI Manager to initiate and control I/O transactions on the SCSI bus.

## The Versatile Interface Adapters

The Macintosh II uses two custom Apple Versatile Interface Adapter (VIA) chips, called VIA1 and VIA2. VIA1 is used mainly to provide control lines for the floppy disk drives and Serial Communications Chip, and to interface the Apple Desktop Bus to the system. VIA2 supports many features, including functions related to interrupts from the NuBus slots, SCSI, and Apple Sound Chip.

## The Apple Desktop Bus

The Apple Desktop Bus (ADB) is a serial communications bus designed to accommodate low-speed input devices. The ADB interfaces to the system through the VIA1 chip. The A/UX kernel provides a set of routines called the ADB Manager. The ADB Manager controls the ADB bus and calls other kernel routines that control the VIA1 chip.

To perform I/O to a device connected to the ADB bus, a driver must control the I/O from the kernel level, to the ADB bus, and to the attached device. A device driver for a device connected to the ADB calls routines in the ADB Manager to control transactions on the ADB bus. For example, the structure of the mouse driver is illustrated in Figure 1-7. The mouse driver calls ADB routines to initiate read operations between the mouse and the ADB.

**Figure 1-6**
The structure of a SCSI disk driver

**Figure 1-7**
The structure of the mouse device driver

## The Serial Communications Controller

Serial I/O is performed through two RS-422 serial I/O ports. The two serial ports are controlled by a Zilog Z8530 Serial Communications Controller (SCC) chip. The serial ports can be used for devices such as printers, modems, and other I/O devices. The SCC chip is controlled in software by the `sccio` driver.

## The Apple Sound Chip

The Apple Sound Chip (ASC) is used with the internal speaker. You can hook up an external mini-phono jack to the external sound connector. The ASC chip is controlled in software by two low-level kernel routines, `sound.c` and `sound.s`.

## The Integrated Woz Machine

The internal floppy disk drives are connected to the system through the Integrated Woz Machine (IWM). The floppy disk driver contains the low-level routines to control the IWM. The floppy disk driver uses these low-level routines to control the floppy disk drive.

## Summary of software drivers and hardware

Figure 1-8 illustrates how an I/O request from a user goes through the kernel, device drivers, low-level routines or managers to reach the actual device. For example, a SCSI device driver calls routines in the SCSI manager to accomplish the I/O on the hardware. Figure 1-9 shows the hardware each device driver interfaces to in greater detail. For example, a SCSI device driver interfaces to the SCSI device through the SCSI bus.

For more specific information on the various hardware interfaces in the Macintosh II, refer to the *Macintosh Family Hardware Reference.*

**Figure 1-8**
Overview of an I/O request from a user program to the hardware

**Figure 1-9**
Overview of the hardware associated with each driver

## Memory-mapped I/O

The Macintosh II uses memory-mapped I/O. This means that each device (peripheral) in the system is accessed by reading from or writing to specific locations in the address space of the computer. Parts of the Macintosh II address space are reserved for performing memory-mapped I/O. Within this reserved address space, specific blocks (addresses) are devoted to each of the hardware interfaces within the computer.

The address space within $5000 0000 to $5FFF FFFF is the area reserved for system I/O address space. All hardware interfaces (except NuBus) are mapped within this address space. The standard NuBus address space is within $F900 0000 to $FFFF FFFF.

By reading from or writing to a location in the system I/O address space or the standard NuBus address space, you are actually accessing (addressing) a particular device.

Each device contains the logic to recognize when it is being addressed. You can use memory-mapped I/O to write to registers on a device or card. Typically only the lowest-level routines directly read from or write to the memory-mapped I/O address space.

By reading or writing to a specific location in memory, you are actually accessing (addressing) a particular device. Illustrations of the address space used in A/UX are shown in Appendix D.

## Interrupt handling by your driver

How your device driver needs to handle interrupts depends on the hardware interface that your device connects to. Apple supplies the low-level software that directly control the hardware interfaces. For a description of these hardware interfaces, refer to the previous section "An Overview of the Hardware". Also refer to Figure 1-4 for an illustration of the interrupt level of each hardware interface.

When a device interrupts, the low-level managers or low-level routines are invoked to handle the interrupt. The low-level routine or manager determines the type of interrupt and what action, if any, to take. For example, if more than one device is connected to that particular hardware interface, the low-level manager might have to poll the hardware to determine which device interrupted.

The low-level routine or manager determines whether or not a higher-level of software (driver) needs to be notified when a device interrupts. Typically, a device generates an interrupt when the device has completed an I/O request. In this case, the higher-level driver responsible for the I/O request needs to be notified that the I/O has completed.

The low-level routine or manager notifies the higher-level driver by calling the interrupt routine of the driver. (The interrupt routine of a driver is also often referred to as the completion service routine.) The interrupt routine of the higher-level driver can then take whatever action is necessary to service the interrupt for the particular device.

For example, if the interrupt is due to I/O completion, the driver usually checks for any error conditions that might have occurred, and takes appropriate actions. A device driver's interrupt routine also typically notifies any user process waiting for the I/O to complete. The synchronization that must exist between higher-level driver routines and the interrupt routine of a driver is explained in detail in following chapters.

If you write a device driver for a SCSI device or ADB device, the driver you create will access your device through one of the low-level managers. Your driver calls a low-level manager to control the hardware interface your device is connected to. When an I/O request completes on a device, the low-level manager is notified of the interrupt.

If you write a slot device driver, the driver you create will access your device through memory-mapped I/O. Your driver can also use the Slot Library to read from slot ROM. Your slot device driver must provide an interrupt routine that will be invoked by the kernel when your slot card generates an interrupt.

As previously described, the low-level routine or manager typically invokes the interrupt routines of higher-level drivers. This means that the low-level routine or manager must obtain a pointer to the interrupt routine of your driver. Before performing I/O to your device, your driver must inform the low-level manager or routine of the address of the interrupt routine of your driver.

Typically drivers call a low-level routine for this purpose during initialization of the device, in either the *driver*`init` or *driver*`open` routines. The following paragraphs briefly describe how to provide the address of your interrupt routine to the SCSI Manager, ADB Manager, and low-level kernel code that manages interrupts from the NuBus.

### Handling interrupts from SCSI devices

To perform I/O on a SCSI device, the driver calls a SCSI Manager routine. The driver passes two parameters to the SCSI Manager routine: the SCSI ID of the device, and a pointer to a request block data structure.

The request block data structure contains a pointer to the interrupt routine of the device driver making the request. This pointer allows the SCSI Manager to associate the driver interrupt routine with a particular SCSI ID. When the SCSI device completes the I/O transaction, the SCSI Manager calls the driver interrupt routine associated with this request on the SCSI ID.

### Handling interrupts from ADB devices

The ADB Manager requires that your driver provide the address of its interrupt routine before any hardware transactions are initiated on the ADB for your device. Your driver should call `fdb_open()`, including as parameters the address of your driver interrupt routine and the ADB address of your device.

The ADB Manager calls this interrupt routine at the end of each ADB transaction to pass back data and to notify the driver that the transaction has completed. The ADB manager also calls the interrupt routine when certain exception device polling conditions exist.

### Handling interrupts from NuBus devices

For NuBus slot card drivers, your driver must tell the kernel the address of the interrupt routine of your driver. You do this at the time your driver is linked into the kernel. To add your driver to the kernel, you create a master script file that specifies how your driver should be linked into the kernel. The master script file for your driver must contain the flags `vs` if your driver receives slot interrupts.

The kernel contains an internal slot interrupt vector table that is used to store addresses of the interrupt routines of each driver that controls a slot. When you specify the flags `vs` in your master script file, the kernel fills in the appropriate entry of this table with the address of your *driver*`int` routine.

After receiving an interrupt from a slot card, the kernel indexes this table by slot number and calls the appropriate driver interrupt routine.

## Where to go from here

After you determine what kind of device you have, the type of device driver to write, (block or character), and the interfaces you need, you are ready to read the rest of this manual. Which chapters you read next depends on the type of device driver you are writing.

### Writing a block device driver

If you are writing a block device driver read these chapters:

#### Chapter 2

This chapter contains kernel programming information that you should read regardless of the type of A/UX driver you are writing.

#### Chapter 3

This chapter describes the routines in a block device driver, data structures used by the kernel and block device drivers, and the buffering the kernel performs for block device drivers.

If you are writing a block device driver for a device that can also be accessed as a character device, read Chapter 4. Pay particular attention to the description of the `physio()` routine.

#### Chapters 9-11

Of these chapters, read the one that describes the hardware interface you are using. These chapters discuss using the NuBus, SCSI, and Apple Desktop Bus.

#### Chapters 12-14

These chapters describe how to add drivers to the kernel. Chapters 12 describes the `autoconfig`(2) utility. Chapter 13 shows a specific example of how to add a device driver to the kernel. Chapter 14 describes how to prepare your driver for distribution to your customers.

#### Appendixes A-G

When writing your driver, use Appendixes A and B as references. Each contains a description, parameters, and error values for the driver and kernel routines discussed in this manual.

Use the other appendixes as needed for your device. For example, Appendix D shows the memory-mapped I/O space used in A/UX.

---

### Writing a character device driver

If you are writing a character device driver, read these chapters:

#### Chapter 2

This chapter contains kernel programming information that you should read regardless of the type of A/UX driver you are writing.

#### Chapter 4

This chapter describes each of the routines a character device driver can provide. The chapter also discusses various methods of buffering that you can implement in your driver.

If you are writing a character device driver that uses a strategy routine, then read Chapter 3, which covers block I/O. Chapter 3 gives background on the use of strategy routines and using kernel buffers.

#### Chapters 5-8

Of these chapters, read the one that applies to the character device driver that you are writing. These chapters discuss three specific implementations of character device drivers: terminal, streams, and network device drivers.

*Chapter 5*

This chapter describes traditional terminal device drivers. The terminal device driver is a special type of character device driver that provides an additional buffering layer to handle terminal I/O operations. Streams terminal device drivers are described in Chapter 7.

*Chapter 6*

This chapter describes streams drivers. The Streams device driver provides a flexible, modular interface for character device drivers. Use Streams drivers in place of traditional character drivers whenever possible.

*Chapter 7*

This chapter describes streams terminal device drivers.

*Chapter 8*

This chapter describes network device drivers. Network device drivers are used for devices that communicate with other machines.

You are not limited to writing terminal, streams, and network character device drivers. You can write a character device driver for other I/O devices, implementing the routines necessary for your device.

### Chapters 9-11

Of these chapters, read the one that describes the hardware interface you are using. These chapters discuss using the NuBus, SCSI, and Apple Desktop Bus.

### Chapters 12-14

These chapters describe how to add drivers to the kernel. Chapters 12 describes the `autoconfig`(2) utility. Chapter 13 shows a specific example of how to add a device driver to the kernel. Chapter 14 describes how to prepare your driver for distribution to your customers.

### Appendixes A-G

When writing your driver, use Appendixes A and B as references. Each contains a description, parameters, and error values for the driver and kernel routines discussed in this manual.

Use the other appendixes as needed for your device. For example, Appendix D shows the memory-mapped I/O space used in A/UX.

# Chapter 2

# The Kernel Programming Environment

This chapter describes kernel features and utility routines of special interest to anyone writing A/UX device drivers. For an overview of A/UX, see the *A/UX System Overview*. For an overview of the A/UX programming environment, see the *A/UX Programming Languages and Tools, Volume I*.

## How a typical I/O request goes through A/UX

This section presents an example that shows the way an I/O request might flow from the user through A/UX to a device. Many of the routines and data structures used in this example are described in detail in later sections. The purpose of this example is to give you an overview of how I/O is accomplished in A/UX.

For example, suppose you wanted to connect a high-speed printer to the SCSI port. You could write a device driver to control this SCSI printer. The following paragraphs describe one possible implementation of such a driver.

The SCSI printer driver in this example is called `prt`. The `prt` driver has the responsibility of printing the user's data to the printer. This printing process involves copying the data to a temporary buffer, translating the data into a format and protocol acceptable for the printer, and controlling the hardware interface.

The `prt` driver contains the code for high-level and device-specific functions, and then calls a SCSI Manager routine to handle the hardware-related tasks of controlling the transaction over the SCSI bus.

Assume the `prt` driver provides the following high level routines accessible through the cdevsw table: `prt_open`, `prt_close`, `prt_read`, `prt_write`, and `prt_ioctl`. In addition the prt driver contains a interrupt routine called `prt_int`.

Assume a user process has already opened this device. The following paragraphs trace a `write(2)` request on the example SCSI printer, from the user request, through the kernel and printer driver, to the device, and from the device back to the user process. Refer to Figure 2-1 for the following discussion.

When a user process issues a `write(2)` on the device file associated with the `prt` driver, the kernel processes the request. The kernel fills out a data structure related to the I/O request. For example, the kernel fills in the fields of the data structure with the number of bytes to transfer and a pointer to the user's buffer.

The kernel uses the major number of the device file to index into the cdevsw table (because this file is a character device file). The kernel calls the routine stored at this index that corresponds to a `write(2)` system call. In this example, the kernel calls `prt_write`, passing the data structure and device number as parameters.

**Figure 2-1**
Trace of a write(2) on the example prt driver

The kernel invokes `prt_write` with the device number and a data structure describing the I/O request. `prt_write` uses a kernel macro to extract the minor number from the device number. `prt_write` checks the minor number to make sure this is a request to a valid device.

The data structure passed to `prt_write` includes a pointer to the user's buffer. Thus, `prt_write` has direct access to the user's data. Because `prt_write` needs to manipulate the user's data, `prt_write` copies the user's data to a temporary buffer.

Next, `prt_write` processes the data, formatting the data according to the requirements of the printer. `prt_write` adds any device-specific protocol, then calls a SCSI Manager request routine to initiate the I/O transaction. One of the parameters to the SCSI Manager routine is a data structure describing the I/O request. For example, this structure includes fields that specify the particular SCSI command, a pointer to the data to transfer, and a pointer to the interrupt routine of the driver making the request.

The SCSI Manager queues the request and returns to `prt_write`. `prt_write` waits for the I/O to complete by issuing a call to `sleep()`. `sleep()` puts the user process to sleep until a corresponding call to `wakeup()` is issued. `sleep()` and `wakeup()` are kernel routines drivers can use to synchronize I/O. They are described in Appendix B.

At this point, the I/O request has reached the hardware. When the hardware finishes the transaction (the requested data has been written to the printer), the SCSI Manager notes which request has completed. The SCSI Manager maintains a data structure that associates requests with higher-level drivers. The SCSI Manager calls the interrupt routine (`prt_int` in this example) of the driver associated with this request.

`prt_int` is the completion service interrupt routine of the `prt` driver. The SCSI Manager calls `prt_int` when a request completes on the printer. The SCSI Manager passes an error code as one of the parameters to `prt_int`. This error code indicates the success or failure of the request. If an error occurred, `prt_int` interprets the error code and decides how to handle the error. If the request was successful, `prt_int` updates the appropriate data structures accordingly and calls `wakeup()`.

The call to `wakeup()` issued by `prt_int` awakens the process that had been waiting on I/O. The call to `wakeup()` will cause `prt_write` to continue to execute from the statement following the call to `sleep()`. `prt_write` sets any error values then returns to the kernel. The kernel sets the return value of the system call and returns to the user process.

This example illustrates that a high-level driver routine is called as a result of a system call on a device file. The driver routine does any necessary processing of the request, and can call other kernel routines or other low-level routines to assist in performing the I/O operation.

When the driver is ready to send the request to the hardware, the driver calls a low-level manager routine to accomplish the I/O on the hardware. If the driver waits for the I/O to complete, the driver must provide an interrupt routine that the low-level manager can call when the request completes. When the request completes, the driver should return any data to the user and return a value indicating the success or failure of the I/O request to the kernel. The return value of the system call indicates the success or failure of the system call to the user process.

## A/UX block and character device drivers

Before writing your device driver, you must first decide what type of device driver to write. The device itself and how it performs in the system will determine the type of device driver you write. The hardware that the device must gain access to will also determine how you write your device driver.

In A/UX there are two types of devices drivers: **block** and **character**. A device driver is called a block or character device driver according to the definitions given next. Also, in some instances, you can write a device driver to be both a block device driver and a character device driver.

Devices can also be classified into two categories: **block** and **character**. These classifications are based on historical definitions; many devices can be considered either a block or character device. Actually the device driver and not the device itself determines whether a device is referred to as a block or character device.

Block device drivers make use of the kernel buffer cache when accessing a device. All data read from or written to a block device is buffered through the kernel buffer cache. Block device drivers are most often used for devices that can contain mounted file systems. The SCSI disk driver is an example of a block device driver.

When a user process reads from a block device, the kernel first checks the buffers in the kernel buffer cache for the requested data. If the data is in the buffer cache, the kernel copies the data from the kernel buffer to the user's buffer.

If the data is not in the buffer cache, the kernel calls the associated block device driver. The block device driver transfers the data from the device to a buffer in the kernel buffer cache. After the block device driver transfers the data to a buffer in the kernel buffer cache, the kernel copies the data to the user's buffer.

When a user process writes to a block device, the kernel copies the data from the user's buffer to a buffer in the kernel buffer cache. Then the kernel invokes the associated block device driver. The block device driver schedules the transfer of data between the kernel buffer and the device,and then returns to the kernel.

Normally the kernel returns to the user, without waiting for the I/O to complete. Thus, `write`(2) requests are usually asynchronous. That is, when the kernel returns from a `write`(2) on a block device, you are not guaranteed that the data has actually reached the device. You are only guaranteed that the kernel has copied the data to the kernel buffer cache and that the block device driver has scheduled the data for I/O.

Character device drivers generally perform I/O asynchronously for a variable number of bytes. Character device drivers can buffer their data in any method as needed. The kernel does not buffer data in the kernel buffer cache for character device drivers as it does for block device drivers. However, because the operation of terminals is important to the system, the kernel does provide many data structures and routines that terminal device drivers can use. Chapter 5 describes terminal device drivers in more detail.

There are functional differences between the various character device drivers. Character device drivers can provide a wide variety of functions and can support many different I/O devices. Examples of character device drivers are printer drivers, terminal drivers, tape drivers, and network drivers.

Some drivers can be written to access the device as either a block or a character device. For example, the SCSI disk driver allows the disk to be accessed as a block or a character device. When the disk is accessed as a block device, data is buffered through the kernel buffer cache. Most I/O to data files access the disk in this manner. When the SCSI disk driver accesses the SCSI disk as a character device, the data is not buffered through the kernel buffer cache, but is transferred directly to the disk. The program `fsck`(1) uses this type of access to repair a damaged disk.

When a block or character device driver directly transfers data between the user's buffer and the device, the driver is often said to be performing **raw I/O**.

Figure 2-2 illustrates various buffering techniques used by block and character device drivers. This figure shows that the kernel buffers data between the user process and the block device in the kernel buffer cache. The kernel is responsible for transferring the data between the user's data area and kernel buffers. Block device drivers are responsible for transferring data between a kernel buffer and the device.

As shown in the figure, character device drivers can directly control the buffering between the user process and the device. The character device driver can implement any buffering techniques necessary to transfer the data to the device. This means the character device driver can either implement its own method of buffering or make use of special kernel data structures, such as `tty` structures, to assist in the buffering of the data.

Remember that, in raw I/O, character device drivers do not have to buffer the data at all. Character device drivers that perform raw I/O usually use a strategy routine similar to a strategy routine used by a block device driver.

Chapter 3 describes block device drivers, and Chapter 4 discusses character device drivers in greater detail.

**Figure 2-2**
Methods of buffering data

## Device files

In A/UX there are three different types of files: regular files (also called ordinary files), device files (also called special files), and pipes. In A/UX, all I/O is accomplished by reading or writing to one of these types of files.

All types of files have an inode (inode refers to index node). Each file has an inode associated with it. **Inodes** are data structures used by the kernel to describe files. The inode of an ordinary file contains information about the file, such as file ownership, access permissions, size of the file, and pointers to the data blocks associated with the file.

The inode of a device file also indicates file ownership and access permissions, but does not contain pointers to any data blocks. This is because device files are used to access devices in A/UX. Instead of pointers to data blocks, the inode of a device file contains the device number associated with the device file.

The **device number** contains the major number and minor number of the device file. The device number is a 16-bit number. The major number is stored in the upper 8 bits and the minor number is stored in the lower 8 bits.

The kernel uses the major number to associate a device with a particular device driver. The device driver uses the minor number to encode information specific to the device. For example, the disk driver uses the minor number to identify a specific logical unit and partition of the disk.

A device file must exist for each device used to perform I/O in the system. You read from or write to a device by reading from or writing to the device file associated with that device. For example, to read the current mouse location, first use `open(2)`, specifying `/dev/mouse` as the device file, then issue your `read(2)` request.

Device files are usually stored in the `/dev` directory. As previously stated, to access a device the device must have a device file. You then use system calls to perform I/O to the device. A device file can be either a block or a character device file.

The A/UX system comes with a set of default device files in the `/dev` directory. You can use these device files to perform I/O on various devices. Device files for new devices are usually created by the startup script of the device driver. The section "Writing Optional Init and Startup Scripts" in Chapter 12 describes how to create a startup script for your device driver.

New device files can be created with the `mknod(1)` command. (You must be superuser to use this command.) For example, to create a character device file for a character device driver with major number 9 and minor number 0, the startup script of your driver could contain the following command:

```
mknod /dev/mydevice c 9 0
```

This command creates the device file `/dev/mydevice` with major number 9 and minor number 0 stored in its inode. You can verify the major and minor numbers for the device file with the `ls -l` command:

```
% ls -l /dev/mydevice

crw-rw---- 1 root root  9,   0       February 29 15:23 mydevice
```

Note the values in the permission field: the first character is either b to indicate a block device file, c to indicate a character device file, d to indicate a directory, or - to indicate an ordinary file. The read, write, and execute permissions are indicated next. Like ordinary files, device files also have permissions associated with them. To read from or write to a device, you must have the proper read and write permissions indicated in the device file for that device.

The superuser can deny access to certain devices by setting the permission field appropriately. For example, the device file `/dev/rdsk/c0d0s31` has the following permissions:

```
% ls -l /dev/rdsk/c0d0s31

crw------- 1 root  root  5,  0  February 29 15:25 c0d0s31
```

Only the superuser or root is allowed to access this section of the disk as a character device.

## Device switch tables

Device switch tables contain an array of device switch structures. Device switch (devsw – pronounced *dev-switch*) structures contain pointers to driver routines that correspond to system calls. These pointers to driver routines are stored in the devsw structure for that device driver. For a user process to perform I/O to a device, the associated device driver must have a devsw structure in the devsw table.

When a user process makes a system call, the kernel uses the major number of the device file to index into the devsw table. The kernel calls the corresponding routine from the devsw structure stored at this index.

The kernel maintains two device switch tables, one for block device drivers and one for character device drivers. These two tables are called the bdevsw (*bee-dev-switch*) and cdevsw (*cee-dev-switch*) tables.

The device switch tables are created whenever a new kernel is generated. Whenever a new kernel is created, including a kernel created by the `autoconfig(1M)` utility, information in the `/etc/master.d` directory is read. This information is used to create the bdevsw and cdevsw tables for the new kernel.

To add your driver to the kernel, you need to write a **master script file** for your device driver in the `/etc/master.d` directory. You provide certain information about your driver in this file: for example, whether your driver is a block or character device driver. The `autoconfig(1M)` utility can then create the appropriate entries in the bdevsw or cdevsw structure for your device driver.

The major number of your device driver is assigned by the `autoconfig(1M)` utility. You create the device file for your device in an init or startup script which you need to write for your device driver. Your init script and startup scripts are passed the major number of your device driver when they are invoked. You can then define the minor number for your device driver and use the major number passed to your init script or startup script to create a device file for your driver. Chapter 12 explains the autoconfiguration process and describes how to create a master script file for your driver.

A device driver that can be used as both a character and block device driver has entries in both the bdevsw and cdevsw tables.You choose which routines corresponding to entries in the device switch structure you need to provide for your device driver.

The kernel gives a device driver all the information it needs to perform an I/O request. The kernel passes this information to the device driver in various parameters.

For example, the kernel passes the device number as a parameter to almost all driver routines. The read and write routines of character device drivers are passed a data structure called a `uio` structure. This structure contains information about the I/O request. Block device drivers receive similar information in a `buf` structure. Chapter 4 discusses the `uio` structure, and Chapter 3 discusses the `buf` structure.

The following sections describe the bdevsw and cdevsw tables in more detail.

## The block device switch table

The **block device switch table** is an array of block device switch structures. The bdevsw structure contains pointers to block device driver routines that correspond to system calls. The bdevsw table is illustrated in Figure 2-3.

The bdevsw table is ordered by the major number for the device. The kernel uses the major number to index into this table. When a user process makes a system call, the kernel calls the corresponding routine from the bdevsw structure stored at this index.

Each block device driver in the system has a bdevsw structure associated with it. The addresses of the driver's open, close, strategy, and print routines are stored in the bdevsw structure for that device. The bdevsw structure is defined in `/usr/include/conf.h` as follows:

```
struct bdevsw {

        int (*d_open) ();

        int (*d_close) ();

        int (*d_strategy) ();

        int (*d_print) ();

} bdevsw[];
```

**Figure 2-3**
The bdevsw table

The *d_open entry and other entries in the bdevsw structure are pointers to routines in the device driver. These routines are responsible for carrying out the I/O request corresponding to the system call. The purposes of these routines are described in the following paragraphs.

d_open is used to prepare the device for I/O. The functions of this routine can include configuring the device, initializing data structures, or setting default settings. If the device does not exist or cannot be made available for I/O, your d_open routine should return an error.

d_close is used to release resources associated with the device. The functions of this routine can include releasing acquired memory, restoring the device to its initial state, or other device-dependent operations.

d_strategy is used to schedule the I/O request for reading or writing. Note that the strategy routine queues the I/O request and then returns to the kernel. The strategy routine does not wait for the I/O request to complete.

d_print can be used to print error messages on the console. Your d_print routine can call the kernel's printf() routine to display the message.

The d_open, d_close, d_strategy, and d_print routines should return a value to the kernel indicating the success or failure of the I/O request. Return values of driver routines are discussed in a following section entitled "Return Values of Driver Routines".

Note: The d_open, d_close, d_print, and d_strategy routines are referred to as the *driver*open, *driver*close, *driver*print and *driver*strategy routines throughout the rest of this manual.

The autoconfig utility initially fills in the bdevsw table with default entries. These default entries in the bdevsw structure can be a pointer to either of the two kernel routines nulldev() or nodev(). The nulldev() routine does nothing, while nodev() returns an error.

If the bdevsw entry contains nulldev() and the user process makes the corresponding system call for that entry, the user process does not receive an error. If the bdevsw entry contains nodev() and the user process makes the corresponding system call for that entry, the user process does receive an error.

Refer to Chapters 12 and 13 for information on how autoconfig(1M) creates and fills in the bdevsw structure for your device.

A sample bdevsw table is shown in Figure 2-4.

```
struct bdevsw bdevsw[] = {
        nodev,  nulldev,        nulldev,        nulldev,        /*  0 */
        nodev,  nulldev,        nulldev,        nulldev,        /*  1 */
        nodev,  nulldev,        nulldev,        nulldev,        /*  2 */
        nodev,  nulldev,        nulldev,        nulldev,        /*  3 */
        nodev,  nulldev,        nulldev,        nulldev,        /*  4 */
        snopen,snclose,         snstrategy,     snprint,        /*  5 */
        nodev,  nulldev,        nulldev,        nulldev,        /*  6 */
        nodev,  nulldev,        nulldev,        nulldev,        /*  7 */
        nodev,  nulldev,        nulldev,        nulldev,        /*  8 */
        nodev,  nulldev,        nulldev,        nulldev,        /*  9 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 10 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 11 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 12 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 13 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 14 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 15 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 16 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 17 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 18 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 19 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 20 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 21 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 22 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 23 */
        hdopen,hdclose,         hdstrategy,     hdprint,        /* 24 */
        hdopen,hdclose,         hdstrategy,     hdprint,        /* 25 */
        hdopen,hdclose,         hdstrategy,     hdprint,        /* 26 */
        hdopen,hdclose,         hdstrategy,     hdprint,        /* 27 */
        hdopen,hdclose,         hdstrategy,     hdprint,        /* 28 */
        hdopen,hdclose,         hdstrategy,     hdprint,        /* 29 */
        hdopen,hdclose,         hdstrategy,     hdprint,        /* 30 */
        nodev,  nulldev,        nulldev,        nulldev,        /* 31 */

        };
```

**Figure 2-4**
A sample bdevsw table

## The character device switch table

The **character device switch table** is an array of character device switch structures. The cdevsw structure contains pointers to character device driver routines that correspond to system calls. The cdevsw table is illustrated in Figure 2-5.

The cdevsw table is ordered by the major number for the device. The kernel uses the major number to index into this table. When a user process makes a system call, the kernel calls the corresponding routine from the cdevsw structure stored at this index.

Each character device driver in the system has a cdevsw structure associated with it. The addresses of the driver's open, close, read, write, ioctl and select routines are stored in the cdevsw structure for that device. The cdevsw structure is defined in `/usr/include/conf.h` as follows:

```
struct cdevsw {

        int       (*d_open)();

        int       (*d_close)();

        int       (*d_read)();

        int       (*d_write)();

        int       (*d_ioctl)();

        struct tty *d_ttys;

        int       (*d_select)();

        struct streamtab *d_str;

}cdevsw [];
```

The *d_open entry and other entries in the cdevsw structure are pointers to routines in the device driver. These routines are responsible for carrying out the I/O request corresponding to the system calls. The purposes of these routines are described in the following paragraphs.

**Figure 2-5**
The cdevsw table

d_open is used to prepare the device for I/O. The functions of this routine can include configuring the device, initializing data structures, or setting default settings, such as the baud rate of the device. If the device does not exist or cannot be made available for I/O, your d_open routine should return an error.

`d_close` is used to release resources associated with the device. The functions of this routine can include releasing acquired memory, restoring the device to its initial state, or other device-dependent operations.

`d_read` is used to read data from a device.

`d_write` is used to write data to a device.

`d_ioctl` is used to perform control operations on a device, to get status from the device, change the configuration of a device, or for other device and driver dependent functions. Driver ioctl routines are commonly used to perform miscellaneous activities, such as rewinding a tape or ejecting a floppy disk.

`d_select` is used to check if I/O has completed or if an exceptional condition has occurred. Select routines are often used to test if a device is ready for reading or writing.

If your device is always ready for reading or writing, the `d_select` entry can point to the `seltrue` routine. `seltrue` is a kernel routine that returns TRUE when invoked as a result of `select`(2) on a device file. If your driver does not provide a `d_select` routine, `autoconfig`(1M) fills in this field of the cdevsw structure with `seltrue` as the default entry.

The `d_open`, `d_close`, `d_read`, `d_write`, `d_ioctl`, and `d_select` routines should return a value to the kernel indicating the success or failure of the I/O request. Return values of driver routines are discussed in the following section entitled "Return Values of Driver Routines".

Note: The `d_open`, `d_close`, `d_read`, `d_write`, `d_ioctl`, and `d_select` routines are referred to as the *driver*`open`, *driver*`close`, *driver*`read`, *driver*`write`, *driver*`ioctl` and *driver*`select` routines throughout the rest of this manual.

In addition to the pointers to the device driver routines, the cdevsw structure has a field for a pointer to a `tty` structure and a field for a pointer to a `streamtab` structure.

If your device driver uses or needs a `tty` structure, then you will want the entry for `d_ttys` defined in the cdevsw structure. Usually only terminal device drivers require a `tty` structure.

If you want the entry for `d_ttys` defined in the cdevsw structure for your device driver, then use the `t` option in your master script file. This instructs the kernel to allocate a `tty` structure and set up a pointer to it in the cdevsw structure for your device driver.

If your device driver uses or needs a `streamtab` structure, then you will want the entry for `d_str` defined in the cdevsw structure. Usually only streams device drivers require a `streamtab` structure.

If you want the entry for `d_str` defined in the devsw structure for your device driver, then use the `s` option in your master script file. This instructs the kernel to allocate a `streamtab` structure and set up a pointer to it in the cdevsw structure for your device driver.

The `autoconfig` utility initially fills in the cdevsw table with default entries. These default entries in the cdevsw structure can be a pointer to either of the two kernel routines `nulldev()` or `nodev()`. The `nulldev()` routine does nothing, while `nodev()` returns an error.

If the cdevsw entry contains `nulldev()` and the user process makes the corresponding system call for that entry, the user process does not receive an error. If the cdevsw entry contains `nodev()` and the user process makes the corresponding system call for that entry, the user process does receive an error.

Refer to Chapters 12 and 13 for information on how `autoconfig`(1M) creates and fills in the cdevsw structure for your device.

A sample cdevsw table is shown in Figure 2-6.

```
struct cdevsw cdevsw[] = {
scopen,   scclose,   scread,   scwrite,   scioctl,
sc_tty, ttselect, 0,                                    /*  0 */
syopen,   nulldev,   syread,   sywrite,   syioctl,
0, syselect, 0,                                         /*  1 */
nulldev,  nulldev,   mmread,   mmwrite,   mmioctl,
0, seltrue, 0,                                          /*  2 */
erropen,  errclose,  errread,  nulldev,   nulldev,
0, seltrue, 0,                                          /*  3 */
nodev,    nulldev,   nulldev,  nulldev,   nulldev,
0, seltrue, 0,                                          /*  4 */
snopen,   snclose,   snread,   snwrite,   snioctl,
0, seltrue, 0,                                          /*  5 */
nulldev,  nulldev,   nulldev,  nulldev,   fpioctl,
0, seltrue, 0,                                          /*  6 */
nulldev,  nulldev,   nulldev,  nulldev,   nulldev,
0, strselect, &disp_tab,                                /*  7 */
mouseopen,mouseclose,mouseread,mousewrite,
mouseioctl, 0, seltrue, 0,                              /*  8 */
nodev,    nulldev,   nulldev,  nulldev,   nulldev,
0, seltrue, 0,                                          /*  9 */
sxtopen,  sxtclose,  sxtread,  sxtwrite,  sxtioctl,
0, sxtselect, 0,                                        /* 10 */
nulldev,  nulldev,   prfread,  prfwrite,  prfioctl,
0, seltrue, 0,                                          /* 11 */
nulldev,  nulldev,   nulldev,  nulldev,   nulldev,
0, strselect, &cloneinfo,                               /* 12 */
nodev,    nulldev,   nulldev,  nulldev,   nulldev,
0, strselect, &shlinfo,                                 /* 13 */
nvram_open,nvram_close,nvram_read,
nvram_write,nulldev,  0, seltrue, 0,                    /* 14 */
nodev,    nulldev,   nulldev,  nulldev,   nulldev,
0, seltrue, 0,                                          /* 15 */
nodev,    nulldev,   nulldev,  nulldev,   nulldev,
0, seltrue, 0,                                          /* 16 */
nodev,    nulldev,   nulldev,  nulldev,   nulldev,
0, seltrue, 0,                                          /* 17 */
nodev,    nulldev,   nulldev,  nulldev,   nulldev,
0, seltrue 0,                                           /* 18 */
nodev,    nulldev,   nulldev,  nulldev,   nulldev,
0, seltrue, 0,                                          /* 19 */
ptcopen,  ptcclose,  ptcread,  ptcwrite,  ptcioctl,
0, ptcselect, 0,                                        /* 20 */
ptsopen,  ptsclose,  ptsread,  ptswrite,  ptsioctl,
0, ttselect, 0,                                         /* 21 */
osmopen,  nulldev,   osmread,  osmwrite,  nulldev,
0, seltrue, 0,                                          /* 22 */
nodev,    nulldev,   nulldev,  nulldev,   nulldev,
0,   seltrue, 0,                                        /* 23 */
hdopen,   hdclose,   hdread,   hdwrite,   hdioctl,
0,   seltrue, 0,                                        /* 24 */
hdopen,   hdclose,   hdread,   hdwrite,   hdioctl,
0,   seltrue, 0,                                        /* 25 */
```

```
hdopen,   hdclose,   hdread,   hdwrite,   hdioctl,
0,   seltrue, 0,                                        /* 26 */
hdopen,   hdclose,   hdread,   hdwrite,   hdioctl,
0, seltrue, 0,                                          /* 27 */
hdopen,   hdclose,   hdread,   hdwrite,   hdioctl,
0, seltrue, 0,                                          /* 28 */
hdopen,   hdclose,   hdread,   hdwrite,   hdioctl,
0, seltrue, 0,                                          /* 29 */
hdopen,   hdclose,   hdread,   hdwrite,   hdioctl,
0, seltrue, 0,                                          /* 30 */
nodev,   nulldev,   nulldev,   nulldev,   nulldev,
0,   seltrue, 0,                                        /* 31 */
nodev,   nulldev,   nulldev,   nulldev,   nulldev,
0, seltrue, 0,                                          /* 32 */

};
```

**Figure 2-6**
A sample cdevsw table

## Return values of driver routines

Your driver routines should return a value to the kernel, indicating the success or failure of the I/O request. For successful requests, your driver routines should return 0 (zero). For unsuccessful requests, your driver routines should return a nonzero positive value that corresponds to an `errno` value. Values for `errno` are defined in the header file `<sys/errno.h>`.

If your driver returns a zero to the kernel, the kernel returns a successful value to the user. The value and meaning of a successful value returned to the user depends on the system call. For example, for successful `open(2)` requests, the kernel returns a positive file descriptor. For successful `read(2)` requests, the kernel returns the number of bytes read.

If your driver returns a nonzero positive value to the kernel, the kernel returns –1 to the user and sets the global variable `errno` according to the value that your driver routine returned.

## Process context and the user structure

In A/UX, a **process** is an instance of a program in execution. When executing a process, the system is said to be executing in the context of the process. When the kernel needs to execute a new process, it does a context switch, and the system executes in the context of the new process. When doing a context switch, the kernel saves enough information about the first process so that it can later switch back to the first process and resume its execution.

Every process has an entry in the kernel proc table. The entry for an individual process is a data structure called the proc structure. The kernel uses proc structures to describe the state of every active process in the system. The proc structure contains all information about the process that is needed while a process is swapped out.

The kernel also maintains information about a process in a data structure called the **user structure** (also called the *u-dot*). The user structure contains all process related information that does not need to be referenced while the user process is swapped out.

One user structure exists for each process in the system. Some of the information kept in the user structure include the program counter (PC) and register values, the process memory management unit (MMU) maps, a pointer to the associated proc structure, and the arguments from system calls. The user structure is defined in the file `<sys/user.h>`.

Whatever process is running at the moment has its user structure mapped at a known location in the kernel address space; processes that are not running have their user structures mapped elsewhere in the kernel. Normally there is only one user structure in the kernel at a time—the process now running. This manual uses the term u-dot to refer to the user structure of the current process.

A device driver should never modify values in the user structure. The kernel gives a device driver all the information it needs to perform an I/O request. The kernel passes this information to the device driver in various parameters.

For example, the kernel passes the device number as a parameter to almost all driver routines. The read and write routines of character device drivers are passed a data structure called a `uio` structure. This structure contains information about the I/O request. Block device drivers receive similar information in a `buf` structure.

## Utility routines and macros

The kernel provides a number of routines that you can use in your driver. This section describes routines that can be called from any device driver; you'll also find additional kernel routines for block device drivers in Chapter 3 and additional kernel routines for character device drivers in Chapters 4, 5, and 6.

Use this section to get general information about kernel routines that can be called. Appendix B is a reference for kernel routines found in this manual. Appendix B provides specific information about the parameters passed to each routine and the error values returned for each routine.

### Setting processor levels

Your driver can set the hardware priority level with the spl*n* routine, which disables interrupts below a specified priority level *n*. Setting the priority level prevents unwanted interrupts from reaching the device. See the `spln` routine in Appendix B for specific values of *n*.

To set the interrupt priority level back to its previous state use the splx(*s*) routine, where *s* is a value returned by the previous `spln` call.

### Waiting for I/O to complete on an address or for an event to occur (sleep)

`sleep()` is used to synchronize I/O by making a process wait (and allowing other processes to run) until a certain event occurs. The event is an address that the calling process passes as a parameter to `sleep()`.

When a driver calls `sleep()`, the kernel changes the process state to "asleep" and removes the process from the run queue. When a process is removed from the run queue because of a call to `sleep()`, the process is often referred to as a "sleeping" process.

When a process's state changes to asleep, a context switch occurs; thus, sleep() should always be called within the process's context.

After the driver calls `sleep()`, the sleeping process will continue to sleep until another routine calls `wakeup()`, using the same address as specified by the process that called `sleep()`.

The `sleep()` routine is passed an address, as just described, and a priority level. Priority levels range from 0 to 127, with 0 having the highest priority and 127 having the lowest priority.

Several processes can sleep on the same address. When more than one process calls `sleep()` with the same address, the priority level determines which routine will execute first.

Signals cannot interrupt processes sleeping at a priority less than the parameter `PZERO`, although they can be swapped out. `PZERO` and `PCATCH` are defined in `<sys/param.h>`.

`PCATCH` is a bit set in the priority level argument to `sleep()` that is OR'ed into the priority field of the proc structure when a driver wants any signals that occur during sleep to be ignored and handled later (for example, page faults and streams processing). If processes sleep at this priority level, `sleep()` will return 0 if awakened or 1 if a signal occurred while sleeping.

## Waiting for I/O to complete on a buffer header (biowait)

The kernel or a driver uses `biowait()` when a process is waiting for a resource called a buffer header, or `buf` structure. The routine `biowait()` is similar to `sleep()`, except a buf structure is always passed as a parameter to `biowait()`. When a driver calls `biowait()`, the kernel sets a flag in the `buf` structure and puts the process to sleep. The process continues to sleep until a corresponding call to `biodone()` is made.

## Notifying a process of I/O completion or an event occurred (wakeup)

The kernel or a driver uses `wakeup()` to notify all processes that are waiting for an event to occur that the event has occurred or to notify all process that are waiting for I/O to complete that the I/O has completed. The event is an address that the calling routine passes as a parameter to the `wakeup()` routine.

All sleeping processes marked with the same address are removed from the sleeping processes queue, placed on a list of available processes, and the process state is changed from "asleep" to "ready to run."

## Notifying a process I/O has completed on a buf structure (bidone)

The kernel or a driver uses `biodone()` to notify a process that I/O has completed on the `buf` structure specified in the call to `biodone()`. All processes sleeping on the `buf` structure are removed from the sleeping processes queue and placed on the ready to run queue.

## Reading from and writing to a user buffer

If you are writing a character device driver you can provide your own method for transferring data between a user buffer and a device. Optionally, you can use various routines provided by the kernel. You can use these kernel routines to copy a single character between the user buffer and a driver buffer, to copy blocks of information between the user buffer and the driver buffer, or to directly copy data between the user buffer and the device.

Your driver can use two routines to read and write a character to and from a user buffer: `ureadc()` and `uwritec()`. `ureadc()` delivers a character to a user buffer associated with a `read(2)` system call. `uwritec()` retrieves a character from a user buffer associated with a `write(2)` system call.

Your driver can use the `copyout()`, `subyte()`, and `suword()` routines to transfer data from a driver buffer to a user buffer. `copyout()` copies blocks of information from the driver buffer to the user buffer, `subyte()` copies a single character from the driver buffer to the user buffer, and `suword()` copies a single integer from the driver buffer to the user buffer.

Your driver can use the `copyin()`, `fubyte()`, and `fuword()` routines to copy data from a user buffer to a driver buffer. `copyin()` copies blocks of information from the user buffer to the driver buffer, `fubyte()` copies a single character from the user buffer to the driver buffer, and `fuword()` copies a single integer from the user buffer to the driver buffer.

Your driver can also use the `uiomove()` routine in place of `copyin()` or `copyout()` to copy data between a user buffer and a driver buffer.

Your driver can use the kernel routine `physio()` to directly copy data between the user buffer and the device. Chapter 4 describes this method of buffering in more detail .

## Gaining access to user address space

To determine whether your driver can gain access to the current user address space memory, call the `useracc()` routine.

## Finding the major number of your device

Your driver can use the macro `major` to find the major number associated with your device. The macro `major` extracts the major number from the device number and returns the major number to the calling routine.

## Finding the minor number of your device

Your driver can use the macro `minor` to find the major number associated with your device. The macro `minor` extracts the minor number from the device number and returns the minor number to the calling routine.

## Encoding the major and minor numbers of your device

You can use the kernel macro `makedev` to encode the major and minor numbers for your device.

## Setting a timeout (timeout)

Your driver can use the `timeout()` routine to set a timer for a minimum number of clock ticks. After the given time period has elapsed, the kernel calls the routine specified as a parameter to `timeout()`. Note that the routine is not called in process context.

`timeout()` can be useful when you want to set a maximum amount of time you are willing to wait for an event to occur. For example, you might want to wait only a certain number of seconds for a device to come online. By using the `timeout()` routine, you could specify that your driver routine be called if the device did not respond after a certain amount of time.

## Removing a Timeout (untimeout)

The `untimeout()` routine is used to remove a timeout previously set by `timeout()`.

If your driver set a timer using the `timeout()` routine and if the timer expires, indicating that the anticipated event did not occur, the routine specified in the call to `timeout()` will be called. If the event does occur before the timer expires, you must call `untimeout()` to cancel the preceding timeout request.

## Delaying execution

Your driver can call `delay()` to make a process wait for a specific interval before resuming execution. `delay()` must be called in process context, because it suspends a process and puts it to sleep for a minimum number of clock ticks. For example, `delay()` is useful in routines that need to wait for a 3.5-inch disk drive to spin up to speed.

## Sending a signal to a user process

**Signals** inform user processes of certain events that occur. For example, your driver may need to send a signal when a modem carrier line drops. The kernel `signal()` routine sends a specified signal to all processes in a process group. `signal()` can be called in any process context.

*Note:* The kernel `signal()` routine is not the same as the `signal(2)` routine, which specifies how the calling process handles signals that are received.

To send a signal to a single process, your driver should call `psignal()`. `psignal()` marks (in the proc structure) that the process should receive a signal and enables the job to run. When a signal is caught in a user process (for example, when the user types a break character), a context switch occurs and the process handles the signal. When a process is executing in the Berkeley signal environment, a signal is not always sufficient to awaken it (for example, if the process is stopped).

# Chapter 3

# Block Device Drivers

This chapter starts with a general discussion of block I/O device drivers and the rules for writing them. This chapter then describes data buffering structures, followed by detailed descriptions of the block device driver's open, close, strategy, and diagnostic print routines. The start and interrupt routines of a block device driver are also discussed.

## Overview

Block device drivers make use of the kernel buffer cache when accessing a device. All data read from or written to a block device is buffered through the kernel buffer cache. Block device drivers are most often used for devices that can contain mounted file systems. The SCSI disk driver is an example of a block device driver.

A block device driver maps logical device block numbers to physical device block numbers. A block I/O logical device is a device consisting of addressable secondary memory blocks. The size of each block is a multiple of the `DEV_BSIZE` constant. (In the past, logical devices have also been called *partitions*.)

The block device driver recognizes the physical devices in the system. The driver's main job is to hide all aspects of the physical device from the kernel and present a logical device interface of $n$ 512-byte blocks, which are numbered from 0 to $n$–1. Thus, to the A/UX operating system, logical devices and physical devices appear to be the same.

Typically, any device with a block I/O driver interface also supports a character I/O driver interface. That is, the source file for the driver contains routines for both block device drivers and character device drivers.

A block device driver can support more than one physical device. In turn, each physical device can contain more than one logical device. Typically, a single physical device, such as a 300-megabyte disk drive, will have a number of logical devices on it.

## Transferring data to and from a block device

After the operating system mounts the file system and opens the device file for the device, a driver reads and writes to a block device in one of two ways:

- indirectly through the kernel buffer cache
- directly through a raw (character) interface

Indirect data transfers take place using the kernel buffer cache. The A/UX kernel provides a cache of buffers to temporarily hold data being transferred between user data space and block I/O devices. Buffered I/O is described in the next section.

Direct data transfers take place using raw I/O. All read and write operations using raw I/O perform input and output directly to and from the device without buffering data. Character device drivers are used to perform raw I/O. Raw I/O is discussed in a later section, and also in more detail in Chapter 4.

## Buffered I/O

Buffered I/O uses two important data structures: the `buf` structure (also called the buffer header) and the `iobuf` structure. Both structures are described in the following sections.

### The buf structure

Each buffer in the buffer cache contains two parts: a `buf` structure and an associated buffer. The **`buf` structure** is a data structure that is used to store control and status information about the buffer. The buffer is a memory array containing disk data. The `buf` structure contains a field (`b_un.b_addr`) that points to the buffer associated with this `buf` structure.

The `buf` structure is the sole argument to the strategy routine of a block device driver. The `buf` structure contains all the information needed to perform the data transfer. The kernel fills out fields of the `buf` structure and then invokes the *driver*`strategy` routine with a pointer to the `buf` structure.

A driver can also use `buf` structures to perform unbuffered or physical I/O, in which case the `b_un.b_addr` field of the `buf` structure points to a portion of user data space.

The buf structure is defined in <sys/buf.h> as follows:

```
struct buf
{
        long    b_flags;
        struct buf *b_forw, *b_back;
        struct buf *av_forw, *av_back;
        long            b_bcount;
        long            b_bufsize;
        short           b_error;
        dev_t           b_dev;
        union   {
        caddr_t         b_addr;
        int             *b_word;
        struct          filsys *b_fs;
        struct          dI-node *b_dino;
        struct          fblk *b_fblk;
        daddr_t         *b_daadr;
        struct          svfsdirect *b_direct;
        } b_un;
        daddr_t         b_blkno;
        long            b_resid;
        struct proc   *b_proc;
        int             (*b_iodone)();
        struct          vnode *b_vp;
        time_t b_start;
};
```

A device driver may need to look at or set the following fields of the buf structure:

- b_flags contains bits that indicate the status of the buffer (B_BUSY flag) and tell the driver whether the device is to be read from or written to (B_READ or B_WRITE flag). When the I/O transfer completes, the driver should set the B_ERROR flag if an error occurred. The complete list of flag descriptions is found in <sys/buf.h>.

- `av_forw` and `av_back` are a pair of pointers that maintain a doubly-linked list of "free" blocks (blocks that can be reallocated for another transaction). A driver can use these lists to link the buffer into driver buffer queues.
- `b_bcount` is the number of bytes to be transferred to or from the buffer.
- `b_dev` holds the device number. The device number contains the major and minor numbers. Your driver can use the kernel macros `major` and `minor` to extract these numbers from the device number.
- `b_blkno` is the device offset (in byte blocks starting at block 0) to be accessed. The constant `DEV_SIZE` is the size of a block.
- `b_resid` is the number of bytes not transferred after the I/O request completes. Your driver should set this field to zero if all bytes were transferred. If an error occurred, your driver should set this field to the number of bytes that were not transferred.
- `b_error` contains a value indicating the success or failure of the I/O request. Your driver should set this field to an `errno` value if an error occurred. If the request was successful, your driver should set this field to 0.

## The iobuf structure

The **`iobuf` structure** is a header for a queue of `buf` structures that are currently involved in I/O operations. Your device driver must declare and initialize one `iobuf` structure for each physical device handled, even if several physical devices use the same device driver. Autoconfiguration can be used to allocate these `iobuf` structures (see Chapters 12 and 13 for details). The `iobuf` structure is defined in `<sys/iobuf.h>` as follows:

```
struct iobuf
{
        int             b_flags;
        struct buf *b_forw;
        struct buf *b_back;
        struct buf *b_actf;
        struct buf *b_actl;
        dev_t           b_dev;
        char            b_active;
        char            b_errcnt;
        struct          eblock *io_erec;
```

```
        int             io_nreg;
        physadr         io_addr;
        struct          iostat *io_stp;
        time_t io_start;
        int             io_s1;
        int             io_s2;
};
```

A device driver interacts with these two fields of the iobuf structure:

- `b_actf` is the first `buf` structure on the `iobuf` queue.
- `b_active` determines whether the device controlled by this `iobuf` is active. If the field is set, an operation is occurring; if the field is 0, no operation is occurring.

When the device is ready for an I/O operation, the driver examines the first buffer on the active queue and sets the `b_active` field. After the operation ends, the driver sets `b_active` to 0, removes the buffer from the active queue, and updates `b_actf` to point to the next buffer.

## The block device driver interface

The following sections briefly describe the routines of the block device driver that are called through the bdevsw table. For a description of how a block device I/O operation occurs, see "Trace of an I/O Request on a Block Device Driver" later in this chapter. Appendix A also includes a description of the parameters, calling sequence, and return values for each of the routines presented in the following sections.

## Opening a block device for I/O

The purpose of the block device driver's open routine is to make sure that the kernel's request to use the logical disk is valid; the *driver*open routine does not actually open an A/UX file. The *driver*open routine of a block device driver is called whenever a user process makes an open(2) system call on a block device file.

### The driveropen routine

The *driver*open routine is used to get the device ready to perform I/O. This process might include initializing data structures and setting the configuration of the device. A block device driver's open routine might also perform other functions:

- Check to see if the device number passed to it as an argument is valid for the physical device. The device number is composed of a major number and a minor number. Your driver can encode the minor number with device or driver specific information. For example, the A/UX disk driver (`hd.c`) encodes the high-order bits of the minor number with the drive number, and uses the low-order bits to index into a table of logical disks for the physical drive.
- Call an optional timer function (if the device's open routine has not been called before) to periodically check the status of the device. For example, your driver could call a routine at specified times to determine if I/O has ended and could reset the hardware if it appears that a hardware problem has occurred.
- Set up addresses or request private data areas for use as long as the device is open. For example, error logging might require a data buffer that stores the number of retry operations.
- Perform device-dependent initialization and status checks to enable the physical hardware to be used, such as waiting for a disk drive to spin up to speed and come on-line.
- Remember that the block device driver can control more than one physical device. For example, a disk controller card may support several physical disk drives and each physical disk drive may have multiple logical disks on it. The *driver*`open` routine must keep track of which physical drives have been previously initialized and opened.

The *driver*`open` routine is called as follows:

```
int driveropen(dev, flag)

dev_t dev;
int flag;

where
```

- *dev* is the device number. The device number contains the major and minor number of the device file. A character device driver should check to see that the minor number passed to it as an argument is valid for the device being called. If not, the driver should return an error.
- *flag* corresponds to the *oflag* parameter specified by the user in the open(2) call. (See `open`(2) in the *A/UX Programmer's Reference.*) Specific values for the *flag* parameter are listed in the `f_flag` field of the file descriptor data structure (in the header file `<sys/file.h>`).
- *driver* is the device prefix.

The *driver*open routine is called with two parameters. The first parameter is the device number of the device file being opened. The *driver*open routine can use the kernel macro `minor` to extract the minor number from the device number. Your driver can encode the minor number with driver specific information. For example, when a driver is used to control more than one device, the minor number is usually encoded to indicate the device or id number of the device.

Your driver can then use the minor number to identify the particular device to which the I/O request is directed. Your driver can also set up arrays indexed by the minor number. Using the minor number in this way lets your driver keep track of which request is associated with a particular device.

After you decide how to encode the minor number for your device and how your driver will use the minor number, remember to create the device file for your device in either an init or startup script.(The init and startup scripts are used with `autoconfig`(1M) and are described in Chapter 12). For example, a driver might use a certain bit in the minor number to select the physical device. Then you would need to create multiple device files in your init or startup script for each different physical device that can be selected.

The *flag* parameter in the *driver*open call corresponds to the *oflag* parameter specified by the user in the open(2) call. The kernel translates the *oflag* values of `O_`*XXXX* into their corresponding flag values of `F`*XXXX*. For example, `O_NDELAY` becomes `FNDELAY`, and `O_RDONLY`, `O_WRONLY` and `O_RDWR` are translated into two flags, `FREAD` and `FWRITE`. The flags of interest to a driver are `FREAD`, `FWRITE`, and `FNDELAY`. The action your driver takes if any of these flags is set is driver dependent. However, your driver does not have to implement actions for any of these flag values. For example, the `O_NDELAY` flag usually has meaning only for terminal devices.

The block device *driver*open routine should report any errors to the kernel by returning a value that indicates the success or failure of the request to the kernel. Your driver should return a zero (0) if the open request was successful. If the open request was not successful, your driver should return a nonzero positive `errno` value to the kernel.

If your driver returns a value indicating success, the kernel returns a file descriptor to the user. The kernel also maintains a count of the number of times this device file has been opened and increments this counter on each successful open(2) call. The kernel uses this information to determine when to call the *driver*`close` routine.

If your driver returns an `errno` value to the kernel, the kernel returns –1 to the user and sets the global variable `errno` to the `errno` value returned by your driver.

## The driverclose routine

The kernel calls the *driver*`close` routine on the last `close`(2) of the block device. If several processes have opened a device, the *driver*`close` routine is called once when the last process that has opened the device closes it.

The kernel maintains a count of the number of times the device file has been opened, and calls the *driver*`close` routine only if this is the last close of the device file.

Note that "called on the last close" actually means that the *driver*`close` routine is called only on the last close of a unique device number. Thus, for a disk that has different device numbers (device files) to represent different partitions on the disk, the *driver*`close` routine will be called each time a partition is closed. Your block device driver needs to make sure that all partitions on a single disk have been closed before performing any final driver close functions.

The *driver*`close` routine is called as follows:

```
void driverclose(dev, flag)
dev_t dev;
int flag;
```

where

- *dev* is the device number.
- *flag* corresponds to the flags from the *oflag* field of the open system call. Specific values for the *flag* parameter are listed in the `f_flag` field of the file descriptor data structure (in the header file `<sys/file.h>`).
- *driver* is the device prefix.

The *driver*`close` routine is used to remove the connection between the user process and the device. The functions of a *driver*`close` routine might include reinitializing driver data structures and device hardware. The *driver*`close` routine should do any necessary processing to make the device available to be opened later.

## Performing I/O (using the strategy routine)

Block device drivers use the kernel buffer cache to move data to and from a physical device. Instead of providing separate read and write routines, a block device driver uses a single `strategy` routine to move data between the buffer cache and a device.

The main functions of the *driver*`strategy` routine are to place the `buf` structure for the I/O request onto the device's active I/O request queue and to call a start routine to begin I/O.

The *driver*`strategy` routine is invoked with a pointer to a `buf` structure. For block device drivers, the kernel fills out all fields in the `buf` structure with information about the I/O request before calling the *driver*`strategy` routine.

The *driver*`strategy` routine is called as follows:

```
void driverstrategy(bp)
struct buf *bp;
```

where

- *bp* is a pointer to the `buf` structure containing information about the I/O request. The `b_un.b_addr` field of the `buf` structure contains the address of the buffer being read or written.
- *driver* is the device prefix.

Your *driver*`strategy` routine uses information in the `buf` structure to perform the I/O request. For example, the `buf` structure indicates the direction to transfer the data, the device the I/O request is directed to, and the number of bytes to transfer.

Your *driver*`strategy` routine should schedule the I/O. This scheduling often involves calling another routine called the *driver*`start` routine. The *driver*`start` routine usually takes care of the low-level details of the I/O transfer, including managing the request queue of buffers waiting to send or receive data.

After scheduling the I/O, your *driver*`strategy` routine should return to the calling routine. Your *driver*`strategy` routine must not issue a call to `biowait()` or `sleep()`. The calling routine has the responsibility of determining whether or not to wait for the I/O request to finish.

---

### Writing to a block device

When a user process writes to a block device, the kernel copies the data from the user's buffer to a buffer in the kernel buffer cache. The kernel fills out a `buf` structure with information about the I/O request. Then the kernel invokes the associated block device *driver*strategy routine, passing a pointer to a `buf` structure as a parameter. The *driver*strategy routine schedules the transfer of data between the kernel buffer and the device, and then returns to the kernel.

After scheduling the I/O, your *driver*strategy routine should return to the calling routine. Your *driver*strategy routine must not issue a call to `biowait()` or `sleep()`. The calling routine has the responsibility of determining whether or not to wait for the I/O request to finish.

For `write(2)` requests, the kernel usually returns to the user without waiting for the I/O to complete. Thus `write(2)` requests are typically asynchronous. That is, when the kernel returns from a `write(2)` on a block device, you are not guaranteed that the data has actually reached the device. You are only guaranteed that the kernel has copied the data to the kernel buffer cache and that the device driver has scheduled the data for I/O.

## Reading from a block device

When a user process reads from a block device, the kernel first checks the buffers in the kernel buffer cache for the requested data. If the data is in the buffer cache, the kernel copies the data from the kernel buffer to the user's buffer.

If the data is not in the buffer cache, the kernel calls the associated block device *driver*`strategy` routine. The *driver*`strategy` routine transfers the data from the device to a buffer in the kernel buffer cache. After the *driver*`strategy` routine transfers the data to a buffer in the kernel buffer cache, the kernel copies the data to the user's buffer.

After scheduling the I/O, your *driver*`strategy` routine should return to the calling routine. The calling routine has the responsibility of determining whether or not to wait for the I/O request to finish.

When the kernel calls *driver*`strategy` as the result of a `read(2)` on a block device file, the kernel usually does wait for the I/O to complete.

The *driver*`strategy` routine can also be used to perform raw I/O. In this case, the character device driver's *driver*`read` and *driver*`write` routines call the kernel routine `physio()`. Parameters to `physio()` include a pointer to a `buf` structure, pointer to a `uio` structure, pointer to the *driver*`strategy` routine, the device number, and a read/write flag.

`Physio()` fills out the `buf` structure passed to it with information specified from the other parameters in the call. Then `physio()` invokes the *driver*`strategy` routine, passing the `buf` structure as a parameter. Raw I/O is further described in a following section.

## The block device start routine

The *driver*`strategy` routine calls another routine provided by the driver called the *driver*`start` routine. The *driver*`strategy` routine calls *driver*`start` to initiate the first I/O operation for a device.

The *driver*`start` routine locates the data on the device from the minor number and block number fields (`b_dev` and `b_blkno`) and uses the buffer address (`b_un.b_addr`) to identify where data should be transferred.

The block device driver maintains a queue of buffers that are being processed for I/O. The *driver*`start` routine places the `buf` structure passed to it on the active I/O queue. If there are no pending requests, the *driver*`start` routine calls lower-level routines to begin the I/O transfer for this `buf` structure. If there are pending requests on the device, *driver*`start` returns to the calling routine.

You can also call the *driver*`start` routine from the driver interrupt routine. The driver interrupt routine is described in the next section.

### The block device interrupt routine

The interrupt routine of a block device driver handles the interrupt generated after the I/O operation is complete. The interrupt routine then calls the *driver*start routine to initiate I/O for the next `buf` on the active I/O queue. The interrupt routine continues to call the start routine to initiate I/O if there is a request to be acted upon and then returns.

When all data is transferred, the driver interrupt routine calls `biodone()` or `wakeup()` to notify any processes waiting for the I/O to complete that the I/O request has finished.

## Trace of an I/O request on a block device driver

Figure 3-1 summarizes the flow of control of an I/O request on a block device driver. The following paragraphs describe how a block device driver processes an I/O request.

**Figure 3-1**
Reading from or writing to a block device

Block device drivers use the kernel buffer cache to move data to and from a physical device. After a user process makes a `read(2)` or `write(2)` system call, a strategy routine is called to move data between the buffer cache and a device. The strategy routine locates the data on the device from the device number and block number fields (`b_dev` and `b_blkno`) of the `buf` structure and uses the buffer address (`b_un.b_addr`) to identify where data should be transferred.

For `read(2)` requests, the kernel searches the buffer cache for the requested block. If the requested block is in the cache, the kernel returns the block immediately to the user program without physically reading the device. If the block is not in the cache,the kernel assigns the block a free `buf` structure and buffer, and then calls the *driver*`strategy` routine to handle the data transfer. If no free buffers are available, the kernel puts the user process to sleep until a buffer is released from another process.

The kernel fills in the `buf` structure with information about the I/O request. The `b_flags` field is set to `B_READ` or `B_WRITE` to indicate the direction to transfer data. The kernel sets the `b_dev` field to the device number. The `b_un.b_addr` field is set to point to the kernel buffer to which data is to be transferred into or out of.

The strategy routine first verifies that the block address (found in the `buf` structure) is valid for the logical device being read or written. If the physical device is divided into several logical devices, the strategy routine must check the requested block to see that it is in the partition specified by the minor number.

The strategy routine places the I/O request on the active queue. The strategy routine then checks to see if the device is busy. If the device is busy, the read must sleep until the device becomes available. An I/O request may be placed in the queue in other than first-in-first-out order. For instance, your driver can search the queue and place the I/O request in an order that reduces disk arm movement. (You can use the `disksort()` routine to order the queue in this manner.)

For `write(2)` requests, the kernel informs the disk driver that it has a buffer whose contents should be written, and the disk driver then schedules the block for I/O. If the disk driver finds a buffer that contains the data, the driver writes the data immediately. Otherwise, the least recently used buffer is reassigned to the write request and the write is performed by marking the buffer as "dirty."

After I/O is complete, the device sends an interrupt to the processor. The driver's interrupt routine is called to remove the `buf` structure from the active I/O queue, to check the queue for more requests, and to call the `biodone()` routine to wake up any sleeping processes. The buffer is placed back on the available list.

The interrupt routine then calls the start routine to start I/O for the next buffer on the active I/O queue. The start routine checks the status of the device, checks and marks the I/O queue for active requests, selects the I/O device, and then calls a command process routine to initiate the I/O process. This interrupt-start mechanism continues until all I/O requests are processed.

## Raw I/O

As previously described, block I/O involves using the buffer cache to transfer data between the user space and the device. This process can be slow, because read and write operations are done a block at a time, and buffer operations such as transferring a block from one buffer to another and flushing out filled buffers must be done.

Your device driver might need to provide the ability to perform raw I/O. This means that data is transferred directly between the device and user address space, without using the data cache. Raw I/O is very useful for backup and restore programs, because your driver can read or write more than one block at a time. For example,a driver can read tape drive files into memory quickly or write tape files onto a magnetic tape cartridge quickly, because the data is input or output in large "chunks."

Your device driver will need to provide entry points in the cdevsw table for *driver*`read` and *driver*`write` routines in order to perform raw I/O.

Your *driver*`read` and *driver*`write` routines can call the kernel routine `physio()` to perform read and write operations for unbuffered I/O. By using `physio()`, you can use `buf` structures and the same strategy routine as used by a block device driver.

Thus, as with buffered I/O, the `buf` structure's device number and block number fields identify where to find data on the disk, and the `buf` structure 's address field identifies where the data should be transferred.

Disks are normally not handled as true block devices. More commonly, they use both the block device and character device (raw I/O) interfaces. For example, Figure 3-2 shows the interface to a typical disk driver. As shown in the figure,the *driver*`read` and *driver*`write` routines referenced by the cdevsw structure are used to perform raw I/O.

**Figure 3-2**
Reading from or writing to a block device using raw I/O

The *driver*`read` and *driver*`write` routines call `physio()`, passing a `buf` structure, `uio` structure, pointer to the *driver*`strategy` routine, device number, and read/write flag as parameters. `Physio()` fills in fields of the `buf` structure. For example, `physio()` sets the `b_un.b_addr` field to point to the user's buffer.

Then `physio()` calls the *driver*`strategy` routine. The *driver*`strategy` routine is usually the same strategy routine invoked as the result of a `read(2)` or `write(2)` on a block device file. The *driver*`strategy` routine queues the request and calls the *driver*start routine to begin I/O.

After the strategy routine returns to `physio()`, `physio()` waits for the I/O to complete by putting the user process to sleep.

When the transfer completes, the driver interrupt routine awakens `physio()` by calling `biodone()`. `Physio()` then updates the `uio` structure and and returns to the *driver*`read` or *driver*`write` routine.

## The diagnostic print routine

The diagnostic print routine of the block device driver can be used to print error messages on the console when device errors occur.

# Performing initialization on a device driver

Your driver can provide an initialization routine called *driver*`init`, where *driver* is the device prefix for your driver. During autoconfiguration, the kernel searches the object file of your driver for a routine with the name *driver*`init`. If the kernel finds such a routine, the kernel adds this information to a list of *driver*`init` routines to call during bootup.

If the kernel does not find a *driver*`init` routine for your driver, the kernel simply proceeds with initialization. You do not have to provide a *driver*`init` routine.

If you do provide a *driver*`init` routine, the kernel will call your *driver*`init` routine during system initialization. However, you can tell the kernel at what stage in the startup kernel code to invoke your *driver*`init` routine.

You do this by using the `p` *opt* flag in your master script file. Some of the options to this command specify whether to call this routine with interrupts disabled or enabled. Refer to Chapter 12 for a complete description of the various options to this parameter.

Typical functions performed in a *driver*`init` routine include initializing hardware, performing diagnostics, and any other code that is needed to make your device available to be used by the system.

## Kernel routines for block device drivers

This section briefly describes kernel routines that your block driver can call to perform specialized functions. See Appendix B for a reference section describing each routine's calling sequence and its parameters and return values. (In addition, Appendix B contains other kernel routines that you can use in a block device driver.)

### Waiting on I/O

The kernel provides two functions for suspending and resuming execution during block I/O transfers: `biowait()` and `biodone()`. (The `iodone()` and `iowait()` routines are defined to refer to the `biodone()` and `biowait()` routines respectively, in `<sys/buf.h>`).

Drivers that have allocated their own buffers and are waiting for data transfer to complete call `biowait()`, which puts the user process to sleep, waiting for I/O to complete on the `buf` structure. The kernel also calls `biowait()` to put the user process to sleep when waiting for `read(2)` requests to complete.

The driver interrupt routine calls `biodone()` to wake up the process or processes waiting on the `buf` structure when the data transfer finishes.

### Buffer routines

You can use these routines to manipulate a buffer in the cache:

- `clrbuf()`—The `clrbuf` macro zeroes the buffer and sets the `b_resid` field of the driver to 0.
- `geteblk()`—The `geteblk()` routine retrieves a buffer from the buffer cache and returns a pointer to the associated `buf` structure to the calling routine. If no `buf` structures are available, `geteblk()` puts the calling process to sleep until one becomes available. Thus, your driver should not call `geteblk()` during interrupt handling.
- `brelse()`—After your driver is finished using a buffer that was previously allocated by `geteblk()`, your driver must call `brelse()` to return the buffer and `buf` structure to the kernel. `brelse()` returns the `buf` structure to the list of free `buf` structures and awakens any processes that might be sleeping on that `buf` structure, or which might be waiting for this `buf` structure.

# Chapter 4

# Character Device Drivers

This chapter describes how to write a basic character device driver. The chapter discusses the various character buffering and control structures first, and then describes the open, close, read, write, ioctl, select, and interrupt routines of character device drivers. For specific information on terminal device drivers and Streams device drivers, see Chapters 5 and 6.

## Overview

Character device drivers control the activity of all those devices that do not buffer their data in the kernel buffer cache. These devices form a large and varied group, and the operations of different devices may require very different device drivers.

You can think of character device drivers as having two or more layers, as illustrated in Figure 4-1. The uppermost layer are those routines accessed through the cdevsw table. These routines might call middle layer routines to handle common functions or to take care of device-specific operations.

For example, a terminal and a printer might share a middle layer of code that performs functions common to both drivers. However, the terminal and printer driver may have different lower layers to handle device-specific operations such as setting the baud rate.

A device driver is not required to have a middle layer of routines. The device driver can contain all the code necessary to process the I/O request, then call low-level routines to initiate and control the hardware operations.

The lowest layer routines are those routines or managers that control the hardware interfaces to the system.

Whenever there is one hardware interface, a single piece of code is used to access all devices. This piece of code might be a manager or a driver. For example, one driver commonly controls all of the serial ports, regardless of which devices are attached to them.

Another example of code that controls a hardware interface is the SCSI Manager. The SCSI Manager controls all accesses to the SCSI port. Higher-level drivers interface to the SCSI Manager, allowing the SCSI Manager to take care of the low-level hardware aspects of controlling transactions on the SCSI bus.

Terminal device drivers can use the `tty` subsystem buffering structures and line discipline routines to handle data buffering in a consistent, structured way. Printers can also use the `tty` structures. These data buffering structures and routines are described fully in Chapter 5, "Terminal Device Drivers."

Streams device drivers are a special implementation of character device drivers. You can implement a terminal device driver as a streams device driver. Streams device drivers also use certain kernel defined data structures. Streams device drivers are discussed in Chapter 6, and streams terminal device drivers are discussed in Chapter 7.

The rest of this chapter focuses on the character device drivers that are not terminal device drivers or streams device drivers.

**Figure 4-1**
The layers of a character device driver

## The character device driver interface

Each character device driver in the system must have a cdevsw structure associated with it. (The cdevsw table is described in "The Character Device Switch Table" section in Chapter 2.) The cdevsw structure contains pointers to driver routines that correspond to system calls.

The cdevsw structures are stored in the cdevsw table. The kernel uses the major number to index into the cdevsw table and calls routines stored in the cdevsw structure at that index. The pointers to driver routines stored in the cdevsw structure are:

- *driver*`open`
- *driver*`close`
- *driver*`read`
- *driver*`write`
- *driver*`ioctl`
- *driver*`select`

where *driver* is replaced by the device prefix for your driver.

Your character device driver must provide routines for each entry in the cdevsw structure according to the needs of your device. For example, a printer device driver usually does not require a *driver*`read` routine. Routines that your driver does not implement are assigned a default entry of either `nulldev` or `nodev` in the corresponding cdevsw structure entry by `autoconfig`.

In addition, your character device driver can provide two other entry points accessible by the kernel:

- *driver*`int`
- *driver*`init`

The *driver*`int` routine is used as an interrupt routine. The *driver*`init` routine is an optional routine your driver can provide to perform initialization functions. These two routines are discussed in the sections "Performing Initialization on a Device Driver" and "Handling Character Device Interrupts".

The following sections describe the character device driver routines with entries in the cdevsw structure that correspond to system calls. Appendix A summarizes the interface each routine must supply, including parameters, calling sequence and return values.

# Preparing a character device for I/O

The kernel calls the character device driver's open routine each time a user program makes an `open(2)` system call on a character device file. The kernel extracts the major number from the device file and uses this number to index into the cdevsw table. The kernel calls the character device driver's open routine stored in the cdevsw structure at this index.

## The driveropen routine

The *driver*`open` routine is used to get the device ready to perform I/O. This preparation might include any initialization not performed by the *driver*`init` routine. Other functions are device dependent, but often include initializing data structures and setting the configuration of the device.

The kernel calls the *driver*`open` routine as follows:

```
int driveropen(dev, flag, ndevp)

dev_t dev,*ndevp;
int flag;
```

where

- *dev* is the device number. The device number contains the major and minor number of the device file. A character device driver should check to see that the minor number passed to it as an argument is valid for the device being called. If not, the driver should return an error value to the kernel.
- *flag* corresponds to the *oflag* parameter specified by the user in the `open(2)` call. (See `open(2)` in *A/UX Programmer's Reference* for a description of *oflag* values.) Specific values for the *flag* parameter are listed in the `f_flag` field of the file descriptor data structure (in the header file `<sys/file.h>`).
- *ndevp* is a pointer to a `dev_t`, which is used in clone open operations for character devices. This parameter is used mainly by streams device drivers.
- *driver* is the device prefix.

The *driver*`open` routine is called with three parameters: *dev*, *flag*, and *ndevp*. The first parameter is the device number of the device file being opened. The *driver*`open` routine can use the kernel macro `minor` to extract the minor number from the device number. Your driver can encode the minor number with driver specific information. For example, when a driver is used to control more than one device, the minor number is usually encoded to indicate the device number or id number of the device.

Your driver can then use the minor number to identify the particular device to which the I/O request is directed. Your driver can also set up arrays indexed by the minor number. Using the minor number in this way lets your driver keep track of which request is associated with a particular device.

After you decide how to encode the minor number for your device and how your driver will use the minor number, remember to create the device file for your device in either an init script or startup script.(The init and startup scripts are used with `autoconfig`(1M) and are described in Chapter 12). For example, a driver might use a certain bit in the minor number to allow the user to select the speed of the output device. Then you would need to create multiple device files in your init or startup script for each different speed setting.

The *flag* parameter in the *driver*`open` call corresponds to the *oflag* parameter specified by the user in the `open(2)` call. The kernel translates the *oflag* values of `O_`*XXXX* into their corresponding flag values of `F`*XXXX*. For example, `O_NDELAY` becomes `FNDELAY`; `O_RDONLY`, `O_WRONLY` and `O_RDWR` are translated into two flags, `FREAD` and `FWRITE`. The flags of interest to a driver are `FREAD`, `FWRITE`, and `FNDELAY`. The action your driver takes if any of these flags are set is driver dependent. For example, if a user specifies `O_RDONLY`, it is up to your driver to decide what a read only request means for your device.

Your driver does not have to implement actions for any of these flag values. For example, the `O_NDELAY` flag usually has meaning only for terminal type devices.

When coding your driver, you need to decide whether you want your device to be an exclusive open device or not. An exclusive open device means only one process is allowed to access the device at a time. For example, tape device drivers are usually exclusive open devices, in order to prevent the data of one user from becoming interwoven with that of another user.

Typically exclusive open devices are implemented in the device driver by setting a flag, for example, `DVROPEN`. When the *driver*`open` routine is called, the driver checks the value of this flag. If the flag is set, another process is using the device. In this case, the *driver*`open` routine refuses to grant access to the new request by returning an error.

If the flag is not set, then another process is not using the device, so the driver sets `DVROPEN`. This process now has exclusive access to the device, until the flag is cleared. The flag is usually cleared by the driver in the *driver*`close` routine.

After your *driver*`open` routine performs any functions required by your device, return a value to the kernel indicating the success or failure of the open request. For example, if initialization did not succeed, you probably want to return an error and refuse to allow the user to gain access to the device.

If your driver returns a nonzero positive `errno` value to the kernel, the kernel returns a −1 to the user, and sets the global variable `errno` to the value returned by your driver.

If your driver returns zero indicating success to the kernel, the kernel marks the file as being open, and returns a file descriptor to the user process. The user process uses this file descriptor in subsequent `read`(2), `write`(2), `close`(2), `ioctl`(2), and `select`(2) calls on this device.

The kernel also maintains a count of the number of times this device file has been opened, and increments this counter on each successful `open`(2) call.

## Closing a character device

After a user process finishes all I/O requests on a device, the user process calls `close(2)` to relinquish access to the device.

The kernel maintains a count of the number of times the file has been opened, and calls the *driver*`close` routine only if this is the last close of the device file. The kernel implements this policy to prevent one user from closing a device while another user is still using the device.

### The driverclose routine

The kernel calls *driver*`close` routine only on the last close of the device file; that is, if no other processes have the device open.

The kernel calls the *driver*`close` routine as follows:

```
void driverclose(dev, flag)
dev_t dev;
int flag;
```

where

- *dev* is the device number.
- *flag* corresponds to the flags from the *oflag* field of the `open`(2) system call. Specific values for the *flag* parameter are listed in the `f_flag` field of the file descriptor data structure (in the header file `<sys/file.h>`).
- *driver* is the device prefix.

The *driver*`close` routine is used to remove the connection between the user process and the device. The functions of a *driver*`close` routine might include reinitializing driver data structures and device hardware. The *driver*`close` routine should do any necessary processing to make the device available to be opened later.

If the device is an exclusive open device, the *driver*`close` routine typically clears any flags that were previously set to indicate the device was open. This clearing of flags allows other processes to gain access to the device.

## Reading from and writing to a character device

The *driver*`read` and *driver*`write` routines of character device drivers are called as a result of the `read(2)` and `write(2)` system calls respectively.

The *driver*`read` and *driver*`write` routines of character device drivers have direct access to the user's buffer. You decide what method of buffering to implement in your character device driver.

The kernel passes two parameters to the *driver*`read` and *driver*`write` routines: the device number and a data structure called the `uio` (user I/O) structure. The `uio` structure describes the data transfer.

Information in the `uio` structure includes a pointer to the user's buffer and the number of bytes to transfer. The kernel fills in the `uio` structure before calling the device driver. The `uio` structure is defined in <`sys/uio.h`> as follows:

```
struct uio {
        struct   iovec *uio-iov;
        int       uio-iovcnt;
        int       uio-offset;
        int       uio-seg;
        int       uio-resid;
};
```

where

- `uio-iov` is a pointer to a buffer containing `uio-iovcnt` number of I/O vectors. Each I/O vector specifies the base (`iov-base`) and the length (`iov-len`) of one transfer.
- `uio-iovcnt` is the number of I/O vectors.
- `uio-offset` is the current offset into the file.
- `uio-seg` is a segmentation flag that can be either `UIOSEG_USER` (user space) or `UIOSEG_KERNEL` (kernel space). This flag is only used by the kernel; your driver can ignore this flag.
- `uio-resid` is initially set to the total size of the transfer request.

The `iovec` structure contains a pointer to the user's data and the number of bytes to transfer. The `iovec` structure is defined as follows:

```
struct iovec{
        caddr_t iov-base;
        int     iov-len;
};
```

where

- `iov-base` is a pointer to the user's buffer associated with this I/O vector.
- `iov-len` is the number of bytes to transfer for the buffer pointed to by `iov_base`.

`Read(2)` and `write(2)` requests use only one `iovec` structure. An array of `iovec` structures are only used in `readv(2)` and `writev(2)` system calls. The system calls `readv(2)` and `writev(2)` allow you to specify more than one buffer in a single read or write request. This process is also referred to as scatter-gather I/O.

In scatter-gather I/O, blocks of data to be written don't have to be contiguous in user memory. Also, when reading from a device into memory, the data comes from the device in a continuous stream, although it doesn't have to be placed in contiguous portions of user memory. A single `iovec` structure is used to describe each contiguous area in memory.

Your driver must keep the `uio` structure updated. Your driver can use `uiomove()` to move data and to update the `uio` structure automatically. Or your driver can use `physio()` to transfer data. `Physio()` also takes care of updating the `uio` structure for your driver.

In addition, the kernel routines `ureadc()` and `uwritec()` can be called to move data one character at a time. If your driver doesn't use `uiomove()`, `physio()`, `ureadc()` or `uwritec()`, your driver must update the `iovec` and `uio` structures.

## The driverread routine

The kernel calls the *driver*`read` routine as a result of a `read(2)` on a character device file.

The *driver*`read` routine is called as follows:

```
int driverread(dev, uio)
dev_t dev;
struct uio *uio;
```

where

- *dev* is the device number.
- *uio* is a pointer to the `uio` structure for the I/O request.

- *driver* is the device prefix.

The kernel invokes the *driver*`read` routine with the device number and `uio` structure as parameters. The driver extracts the minor number from the device number (using the kernel macro `minor`) and typically uses this number to associate the request with a particular device.

The `uio` structure contains all the information the driver needs to know about the I/O request. One of the fields in the `uio` structure contains a pointer to the user's buffer. So the driver can buffer the data according to the requirements of the device, or can directly transfer the data between the user's buffer and the device.

The kernel provides two major routines to assist drivers in performing the I/O operation. Your driver can use the kernel routine `physio()` to directly transfer data between the user's buffer and the device. Your driver can use the kernel routine `uiomove()` to buffer data between the user's buffer and a device. These two routines are discussed in more detail in the sections "Data Transfers using `physio()`" and "Data Transfers using `uiomove()`".

## The driverwrite routine

The kernel calls the *driver*`write` routine as a result of a `write`(2) on a character device file.

The *driver*`write` routine is called as follows:

```
int driverwrite(dev, uio)
dev_t dev;
struct uio *uio;
```

where

- *dev* is the device number.
- *uio* is a pointer to the `uio` structure for the I/O request.
- *driver* is the device prefix.

The kernel calls the *driver*`write` routine with the device number and `uio` structure as parameters. The driver extracts the minor number from the device number (using the kernel macro `minor`) and typically uses this number to associate the request with a particular device.

The `uio` structure contains all the information the driver needs to know about the I/O request. One of the fields in the `uio` structure contains a pointer to the user's buffer. So the driver can buffer the data according to the requirements of the device, or can directly transfer the data between the user's buffer and the device.

The kernel provides two major routines to assist drivers in performing the I/O operation. The kernel routine `physio()` can be used by drivers that directly transfer data between the user's buffer and the device. The kernel routine `uiomove()` can be used by drivers which buffer data between the user's buffer and a device. These two routines are discussed in more detail in the following sections.

## Data transfers using physio( )

Your character device driver can call the kernel routine `physio()` to perform raw I/O (also referred to as physical I/O). The term raw I/O or physical I/O is used to refer to a device driver that directly transfers data between the user's buffer and the device.

You call `physio()` from your *driver*`read` or *driver*`write` routines. `physio()` takes care of many details of the I/O transfer, such as locking the user's buffer into memory, updating the `uio` structure, and unlocking the user's buffer when the transfer is complete.

Your *driver*`read` or *driver*`write` routines can call `physio()` with the following parameters:

```
physio(strat, bp, dev, rw, uio)
int (*strat) ();
struct buf *bp;
dev_t dev;
int rw;
struct uio *uio;
```

where

- *strat* is a pointer to the *driver*`strategy` routine. This usually is the same *driver*`strategy` routine as used by the block device driver for this device.
- *bp* is a pointer to a `buf` structure. The `buf` structure is described in detail in Chapter 3.
- *dev* is a device number that the *driver*`read` or *driver*`write` routine was invoked with.
- *rw* is a flag that indicates the direction to transfer the data.
- *uio* is a pointer to the `uio` structure the *driver*`read` or *driver*`write` routine was invoked with.

physio() takes information from the uio structure, device number, and rw flag and translates it to equivalent information in the buf structure. physio() locks the user's buffer in memory and calls *driver*strategy, passing the buf structure as a parameter. Just as the uio structure fully specifies the I/O request for the *driver*read and *driver*write routines, the buf structure contains all the information the *driver*strategy routine needs to perform the I/O.

The *driver*strategy routine can be the same routine as that used by a block device driver. Refer to Chapter 3 for more information on the functions of a *driver*strategy routine.

The *driver*strategy routine simply schedules the I/O and returns to the calling routine. The calling routine is the kernel when invoked as the result of a read(2) or write(2) on a block device file. The calling routine is physio() when *driver*strategy is invoked as the result of a read(2) or write(2) on a character device file.

The routine that calls *driver*strategy has the responsibility of determining whether or not to wait for the I/O request to complete. physio() always waits for the I/O request to finish by calling biowait(). physio() passes the buf structure as a parameter to biowait(). The call to biowait() puts the user process to sleep until a corresponding call to biodone() is made. Doing this means that when the I/O request completes and your driver interrupt routine is called, your interrupt routine *must* issue a call to biodone() to awaken the user process.

When the transfer completes, your driver interrupt routine should set fields in the buf structure indicating information about the actual data transfer. Your driver interrupt routine should specifically set three fields in the buf structure: b_error, b_flags, and b_resid.

Your driver interrupt routine should set b_error to an errno value and set B_ERROR in the b_flags field if an error occurred in the I/O transfer. Otherwise your driver should set b_error to zero to indicate the I/O transfer was successful.

The b_resid field should be set by your driver interrupt routine to the number of bytes not transferred for the I/O request. If all bytes were transferred, set b_resid to zero.

After setting appropriate fields in the buf structure, your driver interrupt routine should call biodone() to awaken the user process; physio() will then continue to execute. physio() updates the uio structure according to information specified in the buf structure. If the uio structure indicates more data needs to be transferred (only true in the case of a readv(2) or writev(2) system call), physio() again sets up the buf structure and invokes *driver*strategy until all the I/O vectors have been processed.

After the I/O transfer is complete, `physio()` updates the `uio` structure and returns a value indicating the success or failure of the request. `physio()` returns whatever value was specified in the `b_error` field of the `buf` structure. Thus you must be sure your driver interrupt routine sets this value properly. This allows your *driver*`read` or *driver*`write` routine to check the return value of `physio()` and interpret any error value accordingly.

### Using physio() to read from a device

The following paragraphs present an example of the way a character device driver can use `physio()` to accomplish an I/O request. Consider a SCSI tape driver called `tc` (for tape controller). Assume this driver provides the following high level routines accessible through the cdevsw table: `tc_open, tc_close, tc_read, tc_write,` and `tc_ioctl`. In addition the `tc` driver contains an interrupt routine called `tc_ret`. This particular tape driver only allows one request per device.

Assume a user process has already opened this device. This example traces a `read`(2) request on the tape drive, from the user request, through the kernel and tape driver, to the device, and from the device back to the user process. This process is illustrated in Figure 4-2.

When a user process issues a `read(2)` request to the tape, the kernel processes the request. The kernel fills out the `uio` structure related to the I/O request. For example, the kernel fills in the number of bytes to transfer and a pointer to the user's buffer. The kernel uses the major number to index into the cdevsw table and calls `tc_read`.

The kernel invokes `tc_read` with the device number and a pointer to the `uio` structure describing the I/O request. `tc_read` checks the minor number to make sure this is a request to a valid device.

`tc_read` uses a private `buf` structure. This data structure is the same `buf` structure defined by the kernel, but note that this `buf` structure is not associated with the kernel buffer cache. This `buf` structure belongs to the device driver.

`tc_read` calls `physio()`, passing a pointer to `tc_strategy`, a pointer to the `uio` structure, the `buf` structure, the device number, and the `rw` (read/write) flag. `physio()` uses this information to fill in fields of the `buf` structure. For example, `physio()` fills in `b_dev` with the device number, `b_flags` with a value from the `rw` flag, `b_un.b_addr` with the address of the user's buffer as specified in `iov_base` of the `uio` structure, and `b_count` with the length of the I/O transfer, as specified in `iov_len` of the `uio` structure. `physio()` then calls `tc_strategy`.

`tc_strategy` is invoked with a pointer to the `buf` structure that describes the I/O request. `tc_strategy` uses information from the `buf` structure to build the appropriate SCSI command for the read request.Then `tc_strategy` calls a driver start routine,`tc_start`.

**Figure 4-2**
The flow of a read(2) request on the example tc driver

tc_start calls a SCSI Manager routine to start the I/O transaction. The SCSI Manager routine queues the request and returns to tc_start. tc_start then returns to tc_strategy, which in turn returns to physio().

physio() waits for the I/O to complete by issuing a call to biowait(). biowait() puts the user process to sleep until a corresponding call to biodone() is issued. The kernel routines biowait() and biodone() can be used by drivers to synchronize I/O, and are described in Appendix B.

At this point, the I/O request has reached the hardware. After the I/O request has been accomplished (the requested data has been read from the tape drive), the SCSI Manager is notified. When the hardware finishes the transaction, the SCSI Manager notes which request has completed. The SCSI Manager maintains a data structure that associates requests with higher level drivers. The SCSI Manager calls the interrupt routine (tc_ret) of the higher driver associated with this request.

tc_ret is the interrupt routine of the tape driver. The SCSI Manager calls tc_ret when a request completes on the tape drive. The SCSI Manager passes an error code as one of the parameters to tc_ret. This error code indicates the success or failure of the request. If an error occurred, tc_ret interprets the error code and decides how to handle the error. In this case, tc_ret sets b_error to an errno value, sets B_ERROR in b_flags, and sets b_resid accordingly. If the request was successful, tc_ret sets the b_error and b_resid fields in the buf structure accordingly.

After setting fields in the buf structure, tc_ret calls biodone(). The call to biodone() issued by tc_ret awakens the process that had been waiting on I/O. physio() then continues to execute and updates the uio structure from values set in the buf structure. physio() returns the value set in b_error to tc_read. tc_read then finishes any processing and returns a value to the kernel indicating the success or failure of the I/O request. The kernel then returns a value indicating the success or failure of the system call to the user.

## Data transfers using uiomove( )

Your character device driver can call the kernel routine uiomove() to move data between the user's buffer pointed to by the uio structure and a private buffer used by your driver. uiomove() takes care of updating the uio structure, locking and unlocking the user's buffer in memory, and copying the data.

Drivers that need to buffer the data transferred between the user's buffer and a device often call uiomove(). For example, a printer driver that needs to format the data, expanding tabs and other characters, and adding device specific protocol, might call uiomove().

Your driver can call `uiomove()` as follows:

```
int uiomove (address, byte_count, flag, *uio)

caddr_t address;
int byte_count;
int flag;
struct uio *uio;
```

where

- *address* is the address of the buffer where data transfer will occur.
- *byte_count* is the number of bytes to transfer.
- *flag* is either `UIO_READ` or `UIO_WRITE`, indicating whether to copy data into or out of the buffer specified by address.
- *uio* is a pointer to the `uio` structure.

If your driver calls `uiomove()`, you must include as parameters the address of a private buffer belonging to your driver, the number of bytes to transfer, a pointer to the `uio` structure, and a flag indicating the direction to transfer the data.

If your driver specifies `UIO_READ` in the *flag* parameter, data is copied from your driver's buffer to the user's buffer pointed to by the `uio` structure.

If your driver specifies `UIO_WRITE` in the *flag* parameter, data is copied from the user's buffer pointed to by the `uio` structure into your driver's buffer.

To use `uiomove()`, your driver needs a private buffer into which to transfer data into or out of. You can allocate your own storage area in your driver, or you can call the kernel routine `geteblk()` to get a block of memory for your driver.

Your driver can call `geteblk()`, specifying the desired size of memory to allocate. `geteblk()` returns a pointer to a `buf` structure in the kernel buffer cache. The `b_un.b_addr` field of the `buf` structure contains a pointer to the base address of the requested size of memory.

Your driver can call `geteblk()` as follows:

```
struct buf* geteblk(size)
int size;
```

where

- *size* is the requested size of the buffer.

The memory allocated by `geteblk()` is actually a buffer from the kernel buffer cache. `geteblk()` sets the `B_BUSY` flag in the `b_flags` field of the `buf` structure to indicate that the buffer is in use. Doing this gives your driver exclusive access to this buffer.

When you call `geteblk()`, you are really "borrowing" a buffer from the kernel buffer cache. For this reason, when your driver is finished using the buffer, your driver should return the buffer to the kernel buffer cache by calling `brelse()`. `brelse()` is a kernel routine that returns the buffer and `buf` structure to the kernel buffer cache.

Be aware that if no `buf` structures are available, `geteblk()` puts the calling process to sleep `()` until one becomes available. Thus, `geteblk()` must not be called in an interrupt handler.

For a `write(2)`, the *driver*`write` routine first allocates a private driver buffer to hold the data. Most drivers call `geteblk()` for this purpose. The driver then calls `uiomove()` to copy the data from the user's buffer to the driver's buffer. If the driver called `geteblk()`, the driver passes the address in the `b_un.b_addr` field of the `buf` structure as one of the parameters to `uiomove()`. The driver then formats the data in the driver's buffer and sends the data from this buffer to the hardware.

After the hardware accomplishes the write request, the driver interrupt routine should call `brelse()` to return the `buf` structure and buffer previously allocated by `geteblk()`.

For a `read(2)`, the *driver*`read` routine first makes a request to the hardware to read the desired number of bytes of data into the driver's private buffer. Most drivers call `geteblk()` to obtain a buffer to use for the I/O transfer. Then the address of this buffer is given to the hardware as the address to transfer data into.

After the data has been transferred to the driver's buffer, the driver calls `uiomove()` to transfer the data from the driver's buffer to the user's buffer. After the data has been transferred to the user's buffer, the driver should call `brelse()` to return the `buf` structure and buffer previously allocated by `geteblk()`.

## Performing control and miscellaneous functions on a device

The `ioctl(2)` (I/O control) system call provides a general entry point for device and driver specific commands. Your driver can use `ioctl(2)` to allow a process to set hardware device options, software driver options, or other driver dependent functions.

The `ioctl(2)` system call is available for character device drivers only. Block device drivers do not provide a *driver*`ioctl` routine.

Parameters to the `ioctl(2)` system call are a file descriptor, the command to be performed, and an argument to the command. A user process invokes the `ioctl(2)` system call with the following parameters:

```
ioctl (fildes, request, arg)
int fildes, request;
```

where

- *fildes* is a file descriptor returned from a previous `create(2)`, `open(2)`, `dup(2)`, or `fcntl(2)` system call.
- *request* is a command that is driver dependent. The value of this parameter often determines what the user should specify for the *arg* parameter.
- *arg* is the address of an argument associated with the command. The type and value of *arg* is driver dependent. Most drivers pass an address of a structure, allowing various arguments to be specified in different fields of the structure.

For example, to perform an `ioctl(2)` on the console to get the current tty state, you could use the following `ioctl(2)` command:

```
ioctl(fd, TCGETA, &t);
```

In this example, `TCGETA` is an `ioctl(2)` command supported by the driver, and `&t` is the address of a `termio` structure.

Refer to Section 7 of the *A/UX System Administrator's Reference* for a list of commands that individual drivers support in the request field of the `ioctl(2)` system call. You can also look in the header file `<sys/ioctl.h>` for a list of various request parameters.

To use the `ioctl(2)` system call in a user program, you must include the header file `<sys/ioctl.h>` in the code for the user program. Remember that if you are defining new request parameters for your driver, you must include definitions of these values in a header file. In addition, be sure to supply this header file to your users so they can perform `ioctl(2)` system calls on your device.

## The driverioctl routine

The *driver*`ioctl` routine is called as a result of a `ioctl(2)` on a character device file. You can use the *driver*`ioctl` routine to perform control operations on a device, to get status from the device, to change the configuration of a device, or for other device and driver dependent functions. Usually you use *driver*`ioctl` routines to perform miscellaneous activities such as rewinding a tape or ejecting a floppy disk.

The kernel calls the *driver*`ioctl` routine as follows:

```
int driverioctl (dev, cmd, addr, mode)
dev_t dev;
int cmd, mode;
caddr_t addr;
```

where

- *dev* is the device number.
- *cmd* is a command argument indicating the type of operation to be done. The value of *cmd* corresponds to what the user specified in the *request* parameter of the `ioctl(2)` system call. The specific value of *cmd* is driver dependent. You define values for this parameter specific to your driver according to the directions given in a following paragraph.
- *addr* is the address of the arguments to the command. The kernel copies the argument specified by the user into kernel memory and passes this address to the driver. This process allows the driver to copy data freely into or out of the argument in kernel memory space. When the *driver*`ioctl` routine returns to the kernel, if any data is to be returned to the user in the *arg* parameter, the kernel copies the data from kernel memory to the user's buffer.

  The kernel is responsible for copying any data specified by the *arg* parameter between the user's buffer and the driver in `ioctl(2)` system calls. This means the driver does not have to invoke `copyin()` or `copyout()`, although the driver may have to appropriately cast the address passed to it in the *addr* parameter.
- *mode* is an argument that contains values set when the device was opened. The driver can use *mode* to check whether the device was opened for reading or writing.
- *driver* is the device prefix.

The kernel invokes the *driver*`ioctl` routine with the device number, the mode, a command, and an argument that normally serves to pass parameters between a user program and a driver. The *cmd* parameter is defined as follows:

```
#define   cmd_name        aa(x,y,t)
```

where

- *cmd_name* is the name of the command, such as `TCGETA`, `I_PUSH`, `VIDEO_SIZE`, or `MOUSE_BUTTON`.
- *aa* is replaced by `_IO`, `_IOR`, `_IOW`, or `_IOWR`. The macros for `_IO`, `_IOR`, `_IOW`, and `_IOWR` are found in `<sys/ioctl.h>`. The meanings for these values are as follows:

| | |
|---|---|
| `_IO` | No arguments are passed between the user and the driver. |
| `_IOR` | The user reads information from the driver (the driver returns data to the user). |

| | |
|---|---|
| `_IOW` | The user writes information to the driver (the user passes data to the driver). |
| `_IOWR` | Data flows both from the user to the driver and also from the driver to the user. |

- *x* is a unique letter used by your driver to encode the I/O request.
- *y* is a number that distinguishes between various command parameters for your driver.
- *t* indicates the type of the data structure that will be passed in the *arg* parameter in the `ioctl(2)` system call.

For example, the mouse driver encodes one of its *cmd* parameters as follows:

```
#define  MOUSE_BUTTON    _IOR(M,1,unsigned char)
```

This definition says that whenever a user specifies `MOUSE_BUTTON` in the request field of an `ioctl(2)` on the `/dev/mouse` device file, the data structure in the *arg* parameter must be of type `unsigned char`. The `_IOR` indicates data is returned to the user in the *arg* parameter (the mouse driver returns data).

A user program could contain the following code to see whether the mouse button is up or down:

```
#include <sys/ioctl.h>
#include <sys/mouse.h>
#include <fcntl.h>
#include <errno.h>

main()
{

int fd, ret;
unsigned char mousestate;

if ((fd = open("/dev/mouse", O_RDONLY)) == -1)
{
printf("error in open,errno number: %d \n ",errno);
exit(1);
}

if ((ret = ioctl(fd,MOUSE_BUTTON,&mousestate)) == -1)
{
printf(" error in ioctl,errno number: %d \n ",errno);
exit(1);
}

if (mousestate == 0)
       printf("the current mouse state is mouse up \n");
```

```
if (mousestate == 1)
        printf(" the current mouse state is mouse down \n");
}
```

This program first performs an open(2) on the mouse device file. If the request is successful, the kernel returns a file descriptor to the user. The user then performs an ioctl(2), passing the file descriptor, the request name (MOUSE_BUTTON), and an argument as parameters. The mouse driver is invoked with the device number, the command name (MOUSE_BUTTON), and the address of the argument to the command.

The address of the argument is actually a copy in kernel space of the argument specified by the user. This allows the driver to copy the state of the mouse button directly into this area of memory. After the mouse driver returns to the kernel, the kernel copies this data into the argument specified by the user.

The user now has the current state of the mouse button available in the mousestate variable. If mousestate is 0, the mouse button is down. If mousestate is 1, the mouse button is up. Note that the state of the mouse button only applies to the moment when the mouse driver was invoked.

## Checking a device for I/O (select)

A/UX provides the select(2) system call to allow for synchronous I/O multiplexing. A user process specifies which file descriptors to check for their readiness to perform I/O. The user process specifies whether to check each file descriptor for reading, for writing, or for exceptional conditions.

Recall that the select(2) system call is invoked as follows:

```
select(nfds, readfds, writefds, exceptfds, timeout)
int nfds, *readfds, *writefds, *exceptfds;
struct timval  *timeout;
```

where

*readfds*, *writefds*, and *exceptfds* are bit masks where each file descriptor *f* is represented by the bit 1<<*f*.

*nfds* is the number of file descriptors checked, from the bits 0 through *nfds*–1.

*timeout* specifies whether the select(2) call should block or not. If the user specifies a nonzero pointer in this parameter, the pointer points to a timeval structure that indicates the maximum amount of time to wait for the selection to complete. If the user specifies the *timeout* as zero, the select(2) call blocks indefinitely.

A file descriptor is a value returned from a previous `open(2)` call, and corresponds to a particular device file. When a user calls `select(2)`, the kernel calls the *driver*`select` routine associated with each file descriptor. If two or more file descriptors are associated with the same major number of a device, the kernel calls the *driver*`select` routine multiple times, once for each file descriptor.

In addition, if a file descriptor is being selected for more than one function,for example, for both reading and writing, the kernel calls the *driver*`select` routine is called twice: once specifying that the driver check the device for readiness to read, and again specifying that the driver check the device for readiness to write.

`Select(2)` updates each file descriptor mask (*readfds, writefds, exceptfds*) to indicate which file descriptors are ready, based on the value returned by each driverselect routine.

The return value of `select(2)` indicates the total number of ready file descriptors. If the time limit specified in *timeout* expires, `select(2)` returns zero. If an error occurs `select(2)` returns –1 to the user process.

## The driverselect routine

The kernel calls the *driver*`select` routine as follows:

*driver*`select(`*dev, flag*`)`
`dev_t` *dev;*
`int` *flag;*

where

- *dev* is the device number. Your driver can use the `minor` macro to extract the minor number and determine which device the select request applies to.
- *flag* specifies whether to check the device for readiness to read, write, or for an exceptional condition. The paramter *flag* is `FREAD` if the driver is to check if the device is ready for reading. *Flag* is `FWRITE` if the driver is to check if the device is ready for writing. *Flag* has the value zero (0) if the driver is to check for an exceptional condition.

If your device is always ready for reading or writing, you do not have to provide a *driver*`select` routine. The cdevsw entry for *driver*`select` can contain `seltrue`. If `seltrue` appears in the cdevsw entry for *driver*`select`, when a user invokes `select(2)` on the corresponding device, the kernel will return TRUE for that device, by setting the appropriate bit in the file descriptor masks.

## Performing initialization on a device driver

Your driver can provide an initialization routine called *driver*`init`, where *driver* is the device prefix for your driver. During autoconfiguration, the kernel searches the object file of your driver for a routine with the name *driver*`init`. If the kernel finds such a routine, the kernel adds this information to a list of *driver*`init` routines to call.

If the kernel does not find a *driver*`init` routine for your driver, the kernel simply proceeds with initialization. You do not have to provide a *driver*`init` routine.

If you do provide a *driver*`init` routine, the kernel will call your *driver*`init` routine during system initialization. However, you can specify to the kernel at what stage in the startup kernel code to invoke your *driver*`init` routine.

You do this by using the `p` *opt* flag in your master script file. Some of the options to this command specify whether to call this routine with interrupts disabled or enabled. Refer to Chapter 12 for a complete description of the various options to this parameter.

Typical functions performed in a *driver*`init` routine include initializing hardware, performing diagnostics, and any other code that is needed to make your device available to be used by the system.

## Handling character device interrupts

A driver must provide an interrupt routine for handling device interrupts. Exactly how and when interrupts are generated depends upon the device sending the interrupt. For example, each slot device generates only one interrupt for all conditions. Thus, the way your driver handles interrupts also depends upon the device.

How your driver handles interrupts also depends on the lower-level manager or low-level code that your driver interfaces with. For example, to perform I/O on a SCSI device, your driver calls a SCSI Manager routine. One of the parameters to this routine is a pointer to a request block data structure. Your device driver must fill out one of the fields in this structure with a pointer to the interrupt routine of your device driver. Then when the request completes on your device, the SCSI Manager can invoke your driver interrupt routine.

Slot device drivers provide an interrupt routine called *driver*`int`, where *driver* is replaced by the name of your driver. The interrupt routine of a slot device driver is defined during autoconfiguration. To add your driver to the kernel, you create a master script file. You specify the flags `vs` in the master script file to indicate that your driver is a slot device driver that receives interrupts. If you specify the flags `vs` in this file, `autoconfig(1M)` will add the address of your slot device driver interrupt routine to the appropriate entry in the slot interrupt vector table.

When an interrupt occurs on your slot card, the kernel indexes the slot interrupt vector table and calls the routine stored at this address. The kernel passes a single parameter, called `args` (defined in `<sys/reg.h>`) to slot device driver interrupt routines. The kernel fills out various fields of this structure. In particular, the `a_dev` field of the `args` structure contains the slot number of the card that interrupted. This structure allows your driver to determine which of its slot cards interrupted. You can also use the slot number to determine the slot address space for the slot card.

Refer to Chapter 12 for more information on the master script file and the autoconfiguration process. Refer to Chapter 9 for more information on slot device drivers.

# Chapter 5

# Terminal Device Drivers

Terminal devices are special types of character devices that accept, send, and process data from an interactive terminal. They differ from other character drivers in that they perform semantic processing of data and use special routines to buffer data to and from a terminal device.

The A/UX system provides a structured interface to many of the buffering structures and I/O operations of the terminal device driver . This chapter describes the data structures that handle data buffering and shows you how to write terminal device drivers that interface with these structures.

You can also write terminal device drivers as streams device drivers. If you want to write a streams terminal driver, read Chapter 6, "Streams Device Drivers" and Chapter 7, "Streams Terminal Drivers". This chapter focuses on traditional terminal device drivers that do not use streams.

## Buffering and control structures

The buffering structures used to handle data input and output to a terminal device are clists and cblocks, the `ccblock` structure, the `tty` structure, the line discipline, and the `termio` structure.

### The clist and cblock structures

The basic terminal buffering structure is the **clist.** A clist is the head of a linked list queue of character blocks called **cblocks**. Figure 5-1 shows the relationship between a clist and cblocks.

**Figure 5-1**
Clist structure

The `clist` structure is as follows:

```
struct clist {
        int c_cc;
        struct cblock *c_cf;
        struct cblock *c_cl;
};
```

where

- `c_cc` is a count of all the characters in the clist.
- `c_cf` is a pointer to the first cblocks in the clist.
- `c_cl` is a pointer to the last cblock in the clist.

The cblock structure is illustrated in Figure 5-2. Each cblock structure in the clist has the following form:

```
struct cblock  {
        struct  cblock    *c_next;
        char               c_first;
        char               c_last;
        char               c_data[CLSIZE];
);
```

where

- `c_next` is a pointer to the next cblock on the clist.
- `c_first` is a pointer to the first character in the c_data array.
- `c_last` is a pointer to the last character in the c_data array.
- `c_data` is a 64 element character array that stores characters received from or sent to a terminal.

Space for cblocks is allocated at boot time.

**Figure 5-2**
Cblock structure

## The ccblock structure

The ccblock(character control block) structure points to a clist entry. The ccblock is defined as follows:

```
struct ccblock {

        caddr_t         c_ptr;

        ushort c_count;

        ushort c_size;

};
```

where

- c_ptr is a pointer to the character array (c_data) of the cblock.
- c_count is the character count.
- c_size is the size of the character array of the cblock.

Both c_count and c_size are initially set to the size of the cblock character array. c_count is then decreased by the number of characters in the cblock character buffer. The number of characters in the buffer is the difference between c_size and c_count.

## The tty structure

Every terminal device in the system has one **tty structure** (defined in <sys/tty.h>) associated with it. This structure contains information needed to perform terminal I/O. This information includes pointers to the raw, canonical, and output queues; and a pointer to a device driver command processing routine. The tty structure is as follows:

```
#define NCC 8

struct tty {

        struct clist t_rawq;

        struct clist t_canq;

        struct clist t_outq;

        struct ccblock t_tbuf;

        struct          ccblock t_rbuf;
```

```
        int     (* t_proc)();
        ushort t_iflag;
        ushort t_oflag;
        ushort t_cflag;
        ushort t_lflag;
        short  t_state;
        short  t_pgrp;
        char   t_line;
        char   t_delct;
        char   t_term;
        char   t_tmflag;
        char   t_col;
        char   t_row;
        char   t_vrow;
        char   t_lrow;
        char   t_hqcnt;
        char   t_dstat;
        short  t_index;
        unsigned char t_cc[NCC];
        struct proc *t_rsel;
        struct proc *t_wsel;
        struct ttychars t_chars;
};
```

The first three structures, `t_rawq`, `t_canq`, and `t_outq`, are clists. The first clist structure, `t_rawq`, is the raw input queue. The `t_rawq` clist stores raw input data that the terminal's interrupt handler has caught and stored. The second clist structure, `t_canq`, is the canonical queue. This queue stores "cooked" input data, that is, data after the line discipline converts special characters in the raw clist (such as the erase and kill characters) into their canonical forms. These forms are called canonical because the input is processed in a predefined way before it reaches the queue. The third clist structure, `t_outq`, is the output queue used to store data that will be sent to the terminal.

t_rbuf and t_tbuf are ccblock structures. Both t_rbuf and t_tbuf contain pointers to clist entries.The t_rbuf, t_tbuf, t_rawq, t_canq, and t_outq structures are contained in the tty structure, as shown in Figure 5-3.

**Figure 5-3**
Terminal data structures

The `tty` structure fields that are important to someone writing a device driver are as follows:

- `t_rawq`, `t_canq`, and `t_outq` are the raw, canonical, and output queues as just described..
- `t_tbuf` and `t_rbuf` are the device transmit and receive buffers, respectively.
- `t_proc` is the address of the device driver's command processing routine (see "The Driver Command Process Routine" given later in this chapter).
- `t_iflag`, `t_oflag`, `t_cflag`, and `t_lflag` are the input, output, control, and line discipline modes, respectively (see `termio`(7) in the *A/UX Programmer's Reference* for definitions of these modes).
- `t_state` maintains the internal state of the device and the device driver. The state is a composite of one or more bits set in this 16-bit field. The bit definitions are as follows:

| | |
|---|---|
| `TIMEOUT` | A delay timeout is in progress. |
| `WOPEN` | The driver is waiting for an open to complete. |
| `ISOPEN` | The device is open. |
| `TBLOCK` | The driver has sent a control character to the terminal to block transmission from the terminal. |
| `CARR_ON` | This is a software image of the carrier-present signal. It is used with serial chips that supports modem control. For more about this bit, see "Modem Control" given later in this chapter. |
| `BUSY` | Output is in progress. |
| `OASLP` | The processes associated with the device should be awakened when output completes. |
| `IASLP` | The processes associated with the device should be awakened when input completes. |
| `TTSTOP` | Output has been stopped by a CONTROL-S character received from the terminal. |
| `EXTPROC` | A peripheral device is performing semantic processing of data. |
| `TACT` | A timeout for the device is in progress. |
| `CLESC` | The last character processed was an escape character (\). |
| `RTO` | A timeout for a device operating in raw mode is in progress (An example would be if canonical processing is taking place). |
| `TTIOW` | The process associated with the device is sleeping, waiting for the output to the terminal to complete. |

`TTXOFF` Transmission to the terminal is suspended because a CONTROL-S was received from the terminal.

`TTXON` Transmission to the terminal is enabled because a CONTROL-Q character was received from the terminal.

- `t_pgrp` identifies the process group associated with the device. It is used to send signals to the process group.
- `t_line` holds the line discipline type specified in the c_line element of the `termio` structure (a structure that holds values used for `ioctl (2)` operations).
- `t_delct` keeps track of the number of delimiters found while performing semantic processing of data.
- `t_col` records the current column position of the cursor on the terminal.
- `t_row` records the current row position of the cursor on the terminal.
- `t_dstat` can be used by the driver to record driver-defined states.
- `t_cc[NCC]` is an array that holds the control characters specified in the `c_cc` member of the `termio` structure.

## The line discipline

All character devices have a cdevsw structure in the cdevsw table. The cdevsw structure contains pointers to device driver routines corresponding to system calls. The kernel indexes into the cdevsw table and invokes the appropriate device driver routine stored in the character device driver's cdevsw structure.

A terminal device driver is invoked with the same parameters as other character drivers. Once invoked, however, terminal device drivers process the request in a different manner than other character device drivers.

Terminal device drivers use the `linesw` structure, which contains pointers to routines that manipulate character data and buffers. The routines in the `linesw` structure are collectively known as the **line discipline**. After a terminal driver is invoked by the kernel, the terminal driver typically calls a line discipline routine to perform the I/O request.

The linesw structure is defined as follows:

```
struct linesw {

        int (*l_open) ();

        int (*l_close) ();

        int (*l_read) ();

        int (*l_write) ();

        int (*l_ioctl) ();
```

```
        int (*l_input) ();
        int (*l_output) ();
        int (*l_mdint) ();
};
```

The `linesw` structure contains addresses of line discipline open, close, read, write, ioctl, input interrupt, output interrupt, and modem control routines. The line discipline routines maintain the clists, do input preprocessing and output character translation, and perform other terminal services (described in `termio`(7) in *A/UX Programmer's Reference*). The device driver only needs to control the communication line device, and to load and read the device registers.

Line discipline 0 is the system default. The routines for line discipline 0 are as follow:

- `ttopen` Open a terminal device
- `ttclose` Close a terminal device
- `ttread` Read a terminal device
- `ttwrite` Write to a terminal device
- `ttioctl` Perform device-dependent operations
- `ttin` Handle terminal input interrupts
- `ttout` Handle terminal output interrupts

The `t_line` field of the `tty` structure contains the line discipline index into the line discipline switch table. This field can be a value other than 0 (for line discipline 0) if you implement a protocol other than the system default.

## The termio structure

The `termio` structure (defined in `<sys/termio.h>`) holds values used for ioctl operations (such as when the `stty` command calls an ioctl routine to set terminal parameters). It has the following form:

```
#define       NCC     8
struct  termio{
        unsigned short c_iflag;
        unsigned short c_oflag;
        unsigned short c_cflag;
        unsigned short c_lflag;
        char c_line;
        unsigned char c_cc[NCC];
```

```
};
```

where

- `c_iflag` is the input mode of the terminal.
- `c_oflag` is the output mode of the terminal.
- `c_cflag` is the hardware control mode of the terminal.
- `c_lflag` is the local mode of the terminal.
- `c_cline` is the line discipline for the terminal.
- `c_cc` is an array of special control characters.

For the specific values that can be set in these fields, see `termio`(7) in the *A/UX Programmer's Reference.*

## Reading from a terminal

Reading characters from a terminal involves processes both at the user level and the hardware level. Figure 5-4 shows how a character is read from a terminal using the system default, line discipline 0.

**Figure 5-4**
Reading a character from a terminal

When the device hardware receives a character from a terminal, it interrupts the CPU, causing the device driver interrupt function to be entered. The character driver interrupt routine services the device hardware and transfers characters from the device to the receive buffer (`t_rbuf`) of the device's `tty` structure. Each character is checked for validity (parity), and start and stop characters (CONTROL-Q and CONTROL-S). If an invalid character is found, the read interrupt routine must take appropriate action, such as aborting the character transmission or asking for retransmission. It then calls the line discipline 0 input interrupt function, `ttin`, to transfer characters from the receive buffer to the raw queue (`t_rawq`). `ttin` also copies characters from the receive buffer into the transmit buffer (`t_tbuf`) and calls `ttxput` to echo them to the screen.

If the number of characters in the raw queue exceeds a level called the **high-water mark**, `ttin` calls the device driver command process routine to send a stop character to the device to suspend input until the number of raw queue characters falls below a **low-water mark.** High-water marks vary according to the baud rate. (The ratio of the high-water mark to the low-water mark is roughly 9 to 1.)

By suspending input, other processes can get blocks. When the raw queue character count exceeds 256 characters, `ttin` flushes the terminal input queues. If an *stty* character is found (see `stty`(1) for a description), `ttin` sends the appropriate signal to the process group associated with the device. If processes associated with the device are sleeping (during a call to `ttread`) and `ttin` finds a delimiter character, `ttin` awakens the sleeping processes. The `ttin` function also takes care of echoing the characters input back to the terminal by putting them in the output queue as they arrive.

When the terminal is operating in raw mode, the `tty` structure contains the number of characters needed and the amount of time waited before processes associated with the device are awakened. If the minimum character count has been met, `ttin` awakens processes associated with the terminal. If the character count has not been met and a time has not been specified, `ttin` calls `timeout` to awaken the sleeping process after the time period specified.

After a user program calls the `read`(2) system call, the line discipline read routine, `ttread`, is called after a user has typed in a character. `ttread` first transfers the characters from the raw queue to the canonical queue and calls the canon routine to perform canonical processing of data as characters are transferred. If no characters are available, it sleeps on the address of the raw queue until characters become available. To do this, `ttread` checks if there are characters on the canonical queue. If no characters are found, `ttread` places characters from the raw queue onto the canonical queue. This process continues until the number of characters requested has been transferred (and if no errors occur). If a delimiter is found, the routine takes characters from the canonical queue and calls `copyout()` to move them to the user data space.

Before returning, `ttread` checks to see if input is blocked. If data transmission from the terminal has been blocked because the number of characters in the raw input queue exceeded the high-water mark, and if the read has reduced the number of characters to below the low-water mark, `ttread` calls the device driver command process routine to resume transmission from the terminal.

## Writing to a terminal

Writing characters to a terminal involves the output queue (`t_outq`). A transmit buffer is used to buffer characters that will be written. Figure 5-5 shows how a character is written to a terminal using the system default, line discipline 0.

**Figure 5-5**
Writing a character to a terminal

After a user program makes a `write`(2) system call, the terminal driver write routine is called, which in turn calls the line discipline write routine. The line discipline 0 write routine is called `ttwrite`; this routine moves the characters to be sent to be output from the user data space to the output queue and calls `ttxput` to output the contents of the transmit buffer to the terminal. If the output buffer is empty, the line discipline output routine is called to move characters from the output queue to the buffer.

After a character is printed on the screen, an interrupt is generated that causes control to be passed to to the driver transmit interrupt handler. This interrupt indicates that the terminal is ready to accept another character for transmission. If the device doesn't generate transmit data interrupts, this routine should pause for as long as it takes a character to be transmitted between each character transmission. The driver write interrupt routine gets the characters from the transmit control buffer and places them into the device transmit register to output the next character. The driver then sends the next character in the transmit buffer to the device. The line discipline output interrupt routine is called to refill the transmit buffer with characters from the output queue.

## The parts of a terminal device driver

The cdevsw routines found in other character device drivers are also found in a terminal device driver. (See Chapter 4 for general information about character device drivers). Unlike other character drivers, however, terminal drivers must provide pointers to line discipline routines that perform terminal-specific operations. These routines are described next.

### The open routine

The open routine of the terminal device driver is invoked with two parameters: the device number and a flag value. Chapter 4 describes the general functions of a driver open routine.

The terminal device driver open routine calls the following line discipline open routine:

```
(*linesw[tp->t_line].l_open)(tp);
```

`tp->t_line` is an index into the linesw table. The routine pointed to by the `l_open` entry in the linesw structure at this index is invoked.

The line discipline routine establishes a connection between a process and a device, allocates a cblock for the receive buffer of the `tty` structure and calls a driver command process routine with arguments `tp` and `T_INPUT`.

## The close routine

The close routine of the terminal device driver is invoked with two parameters: the device number and a flag value.Chapter 4 describes the general functions of a driver close routine.

The terminal device driver close routine calls the following line discipline close routine to close a device:

```
(*linesw[tp->t_line].l_close)(tp);
```

`tp->t_line` is an index into the linesw table. The routine pointed to by the `l_close` entry in the linesw structure at this index is invoked.

This line discipline routine transmits any characters in the transmit buffer (`t_tbuf`) to the terminal, clears all `tty` buffers and queues, resets the `ISOPEN` bit in the `tty` structure passed to it as an argument, and returns all used cblocks to the list of free cblocks. After calling the driver close routine, the terminal link disconnects and control returns to the calling program.

## The read routine

The read routine of the terminal device driver is invoked with two parameters: the device number and the `uio` structure. The line discipline routines update the `uio` structure for the terminal driver, and take care of many other aspects of performing the I/O.

The terminal device driver read routine calls the following line discipline read routine:

```
(*linesw[tp->t_line].l_read)(tp, uio);
```

`tp->t_line` is an index into the linesw table. The routine pointed to by the `l_read` entry in the linesw structure at this index is invoked.

This line discipline routine performs canonical processing upon raw queue data, and then transfers the data to the canonical queue. After processing, data is transferred from the canonical queue to user data space.

## The write routine

The write routine of the terminal device driver is invoked with two parameters: the device number and the `uio` structure. The line discipline routines update the `uio` structure for the terminal driver, and take care of many other aspects of performing the I/O.

The terminal device driver write routine calls the following line discipline write routine:

```
(*linesw[tp->t_line].l_write)(tp, uio);
```

`tp->t_line` is an index into the linesw table. The routine pointed to by the `l_write` entry in the linesw structure at this index is invoked.

This line discipline routine transfers characters from user data space to the output queue as long as the high-water mark isn't exceeded. As characters are put on the output queue, processing is done to expand tabs, and add delays for newline, carriage return, and backspace characters. When the high-water mark is reached, the routine sleeps on the output queue address. The line discipline write routine then calls the driver command process routine to initiate or resume output to the device.

## The ioctl routine

The device driver ioctl routine normally calls the line discipline routine `ttiocom` with the same arguments that the driver's ioctl function was called with.Driver ioctl routines set parameters related to buffering and character processing. Two `ioctl`(2) commands, `TCGETA` and `TCSETA`, are used to set up terminal characteristics in the `termio` structure and send these commands to the device. For example, your driver can enable the CONTROL-S and CONTROL-Q keys and set characters for erasing lines and interrupting programs. When your driver calls an ioctl routine, it is passed a pointer to a `termio` structure that the line discipline uses to read in the terminal parameters and to set up the terminal.

## The input and output interrupt routines

After receiving an input interrupt, the device interrupt routine calls the line discipline input interrupt routine to process newlines, carriage returns, and uppercase characters (as specified in the `tty` structure); to place the converted characters in the raw queue; and to echo characters to the screen. The input interrupt routine also calls the driver process control routine to stop or restart input from the device, if necessary.

The line discipline write routine calls the line discipline output interrupt routine to move characters from the output queue to the transmit buffer. This routine implements the actual timing delays needed during output. After detecting a delay in the output queue, the routine calls the kernel `timeout()` function to arrange for an entry after a specified time period has elapsed. This delayed entry invokes the driver command process routine to resume output.

## The modem interrupt routine

This routine is currently unsupported.

## The driver command process routine

The device driver must provide a command process routine (also called the `proc` routine) to process device-dependent operations. The `t_proc` member in the `tty` structure points to the command process routine for the line discipline routine that was initialized when the device was opened. The command process routine has the following format:

*prefix*`proc(`*tp, cmd*`)`

```
struct tty *tp;
int cmd;
```

where

- *prefix* is the device prefix.
- *tp* is the address of the device's `tty` structure.
- *cmd* is an integer command, as described next.

The commands are defined in <sys/tty.h>. For line discipline 0, *cmd* can be one of the following:

| | |
|---|---|
| `T_OUTPUT` | Checks to see if the `t_state` member of the `tty` structure is busy or suspended. If so, `T_OUTPUT` does nothing. If `t_state` is not busy, the transmit control block is checked and, if empty, `T_OUTPUT` calls the line discipline output interrupt routine to move characters from the output queue to the transmit control block. A character is then output (if not done by the driver transmit interrupt routine) or `t_state` is set to `BUSY`. |
| `T_TIME` | Notifies the driver that delay timing for a break, carriage return, or other character has completed. This command makes sure that a break signal is not sent to the device and falls through to `T_OUTPUT`. |
| `T_SUSPEND` | Suspends output to the terminal (that is, a CONTROL-Q character has been received). `T_SUSPEND` sets the `t_state` member of the `tty` structure to `TTSTOP`. `T_SUSPEND` is called when a user program invokes `ioctl`(2) with the command argument `TCXON` and the third argument equal to 0. |

| | |
|---|---|
| T_RESUME | Resumes output to the terminal. T_RESUME is called when a user program calls ioctl(2) with the command argument TCXON and the third argument equal to 1. Both T_RESUME and T_WFLUSH fall through to T_OUTPUT. |
| T_BLOCK | Blocks further input when the input queue reaches the high-water mark. T_BLOCK turns off TTXON and turns on TTXOFF and TBLOCK in t_state. |
| T_UNBLOCK | Allows further input when the input queue falls below the high-water mark. TTXOFF and TBLOCK are reset. |
| T_RFLUSH | Resets TTXOFF and TBLOCK if TBLOCK is set; otherwise, T_RFLUSH does nothing. The purpose of T_RFLUSH is to flush pending input (if any). |
| T_WFLUSH | Clears all characters from the transmit buffer. |
| T_BREAK | Sends a 0.25 second break to the device. T_BREAK sets TIMEOUT in t_state and calls timeout with a value of ttrstrt as the function argument. T_BREAK is called when a user calls ioctl(2) with TCSBRK as the command argument and 0 as the third argument. |
| T_INPUT | Prepares a device to receive input. T_INPUT is called by the line discipline 0 ioctl routine when the line discipline changes. The command processing routine makes sure that the device can accept input. |
| T_PARAM | Notifies the driver that the device parameters have changed and that the parameter setting routine should be called to change hardware settings. |

## Modem control

Modem control is an optional feature that allows a driver to acknowledge signals on a serial line. Normal terminal operations occur on a direct connect line where the carrier signal is unimportant. For modem operations, such as for a dial-in line, a driver must be able to detect changes in the carrier signal.

For modem control to exist, the serial controller hardware must support the feature. If your system's serial board generates a modem control interrupt, a drop in the carrier detect is easily seen as a hang-up. For boards without modem control interrupts, the driver must use timeouts to poll the device for state changes.

To accomplish modem control,the following ioctls are provided. Note that not all devices support any or all of them. UIOCTTSTAT is always supported for those devices that support modem control.

`UIOCTTSTAT` This ioctl returns 3 bytes. The first byte is 1 if `UIOCMODEM` is enabled and is 2 if `UIOCEMODEM` is enabled. The second byte is 1 if `UIOCDTRFLOW` is enabled. The third byte is 1 if `UIOCFLOW` is enabled.

The default is `UIOCMODEM/UIOCNOFLOW`. These ioctls are "remembered" when a device is closed,and then reopened. The following four ioctls are mutually exclusive. (Here DCD is the input and DTR is the output).

`UIOCMODEM` Modem control (DTR/DCD) is enabled. DCD is required before a device can be opened. If removed, the device is "hung up"; upon opening, DTR is asserted.

`UIOCNOMODEM` Modem control is not enabled. DTR is still asserted, but DCD is ignored and device open operations always complete without waiting.

`UIOCDTRFLOW` DCD (on some printers this is the DTR line) is used for flow control. DCD must be asserted before characters can be transmitted.

`UIOCEMODEM` European-style modem control (`DTR/DCD/RI`) is enabled. DCD is required before a device can be opened. If removed, the device is "hung-up"; upon opening the device, DTR is not asserted until an RI input is detected.

The following ioctls are also included. In these ioctls, RTS is the output and CTS is the input. They are mutually exclusive.

`UIOCNOFLOW` Hardware flow control is disabled. RTS is asserted before transmitting data (or it is asserted continuously). CTS is ignored.

`UIOCFLOW` Hardware flow control is enabled. RTS is asserted before transmitting data. CTS must be asserted by the other end before transmission can begin (which is required for every character).

# Chapter 6

# Streams Device Drivers

In this chapter, you'll learn how a stream passes information from a user process to a device. You'll also learn about parts of a stream, Streams modules and drivers, and the data structures needed to operate in a Streams environment. This chapter is not intended to be a complete reference for all Streams tools and facilities—rather, you should use it as an introduction to the most important features of streams drivers. Before you write a Streams device driver, you should read the *Streams Programmer Guide* by AT&T.

To help you write Streams-based terminal drivers, A/UX provides the `ttx` library, a set of kernel support routines. With this library, writing a Streams terminal driver is similar to writing a traditional character device driver. You can find details about this library in Chapter 7. For a list of differences between AT&T's System V Release 3 Streams and the version supported by A/UX (System V Release 2.1), see Appendix F.

## What is Streams?

A **stream** is a full-duplex processing and data transfer path between a driver in kernel space and a process in user space. **Streams** is a collection of system calls, kernel resources, and kernel utility routines that can create, use, and dismantle a stream. Streams defines standard interfaces for character input/output within the kernel, and between the kernel and the rest of the A/UX system. To implement these interfaces, a set of system calls, kernel resources, and kernel routines are provided.

By having a standard interface and mechanism, drivers can be modular and portable with easy integration of high-performance network services and their components. A set of library routines and facilities provides buffer management, flow control, scheduling, multiplexing, and asynchronous operations of streams and user processes. One advantage of Streams drivers is that you can insert modules into a stream to process data that passes between a user process and the driver. Streams is upwardly compatible with the character I/O user interface; thus, it's better to write Streams drivers instead of standard character drivers.

## Parts of a stream

A stream has three parts:

- a stream head
- optional modules
- a stream-end (which contains the driver)

Data in a stream is said to travel downstream from the stream head to the stream end or upstream from the stream end to the stream head. Streams passes data through a stream in the form of **messages**, which are linked message blocks consisting of data structures and a buffer block.

A stream is shown in Figure 6-1.

**Figure 6-1**
View of a stream

The **stream head** provides the interface between the stream and the user process. Its main function is to process Streams-related user system calls. It is an integral part of the kernel.

A **module** processes data that travels between the stream head and driver. A stream can contain zero or more modules, each of which is associated with two queue structures (described later in this chapter).

The **stream end** is the part of the stream closest to the external device interface. The stream end contains the Streams driver, which is a special type of module.

## Building a stream

A stream is initially constructed when a user process makes an open(2) system call referencing a Streams special file. This call causes a kernel resident driver to be connected with a stream head to form a stream. Subsequent `ioctl`(2) calls select kernel resident modules and cause them to be inserted into the stream.

The first step in building a stream is creating a minimal stream containing a stream head and a Streams driver. This step takes place by allocating and initializing head and driver structures (which is done automatically when the Streams driver is opened, linking modules to form a stream and calling the driver open routine). The second step in building a stream is to add optional modules, if any, to the stream. (Another term for adding a module is **push**; removing a module is known as a **pop**.) Modules are added in last-in-first-out order.

# Streams modules and drivers

A **Streams module** is a pair of queues that are used to perform intermediate processing on messages flowing between the stream head and the driver. One queue is used to perform functions on messages passing upstream through the module, and the other queue is used to perform functions on messages passing downstream through the module. A module can function as a communication protocol, a line discipline, or a data filter.

A **Streams driver** is the stream end, which is the closest end to the external device interface. A Streams driver can be a device driver or a software driver called a *pseudo-device driver*. Like a module, a driver is composed of two queues, but a driver has additional attributes in a stream and in the operating system. The principal functions of a device driver are device handling, and transforming data and information that pass between the external interface and a stream.

There are two significant differences between modules and drivers. First, a device driver must be accessible from an interrupt and from the stream. Second, a driver can have multiple streams connected to it. Multiple connections occur when more than one minor device uses the same driver. Drivers occupy a file system node and can be opened like any other device. Modules, on the other hand, don't occupy a file system node, but are identified through a separate naming convention and are inserted into a stream in last-in-first-out order. Because modules aren't associated with processes, they can't gain access to information in the u-dot. The only system calls that modules and drivers can interact with directly are open(2) and `close`(2).

## Data structures

The following data structures provide the Streams driver interface to the operating system:

- `streamtab`
- `qinit`
- `module_info`

These need to be set up once for each driver (not once for each device). They refer to each other as well as to the routines that are called to perform the various stream functions. The streamtab data structure must be declared external because it is referenced externally and all the data structures are accessible from it. The other data structures are declared static.

The `streamtab` structure contains pointers to the driver's read and write `qinit` structures. The `qinit` structure contains a pointer to the put, service, open, and close procedures. `module_info` contains a pointer to the processing procedures.

## Messages

Streams passes data between a driver and the stream head in the form of messages. A message consists of one or more message blocks. These message blocks can be linked and placed in a message queue. When several message blocks make up one message, the type of the first block determines the message type and contains links to the preceding and next message blocks.

Streams maintains its own message storage pool. Messages are allocated as single blocks, each of which contains a data buffer of a certain size. If processing causes the data in a message to exceed the buffer size, the procedure can allocate a new message containing a larger buffer for it, or it can allocate a new message that holds the new data and links the two messages together. Use the `allocb` utility to allocate message storage from the Streams pool . (These utility routines are described in AT&T's *Streams Programmer Guide*). This utility returns a message block containing a buffer of the size requested (or larger) or `NULL`, if the request fails. You can specify the level of message pool priority (`BPRI_HI`, `BPRI_MED`, and `BPRI_LO`) to let you better allocate Streams memory resources.

When dealing with messages and message queues, a driver should always use the Streams utility routines described later in this chapter. To make it easier to push modules arbitrarily on the stream, modules shouldn't require the data in an `M_DATA` message to follow a particular format, such as a specific alignment. A module shouldn't change the contents of a data block referenced by other modules. Use the `copymsg` utility to copy the data to a new block.

## Message types

Each messages has a defined message type that identifies the contents of the message. The **message type** is a defined set of values identifying the contents of a message block and message. Modules and drivers can generate most of these message types. There are two levels of message queuing priority: priority and ordinary. When a message is queued, the `putq` utility places priority messages first-in-first-out at the head of the message queue. Priority messages are not subject to flow control, so their associated queue is always scheduled. Ordinary messages are placed in the message queue after priority messages.

The most commonly used types are as follows:

- `M_DATA` contains ordinary data.
- `M_PROTO` contains internal control information and associated data.
- `M_PCPROTO` is like `M_PROTO`, except for priority differences and additional attributes.
- `M_IOCTL` contains an ioctl request.
- `M_IOCACK` and `M_IOCNAK` contain a reply from an ioctl.

(For a complete list and descriptions of all the message types, see AT&T's *Streams Programmer Guide*.)

`M_DATA` messages are generally sent bidirectionally on a stream, and their contents can be passed between a process and the Stream head. The `allocb` routine creates `M_DATA` messages by default. (For more information, see"Utility Routines" given later in this chapter.)

M_PROTO and M_PCPROTO messages carry service interface information among modules, drivers, and user processes. These messages are sent bidirectionally on a stream and their messages can be passed between a process and the stream head. An M_PROTO message block typically contains implementation-dependent control information. The contents of the first message block is the control part, and any following M_DATA message blocks are the data part. M_PCPROTO has the same format and characteristics as M_PROTO, but is called a priority message and is not subject to flow control. This means that when an M_PCPROTO message is placed on a queue, its service routine is enabled. Only one M_PCPROTO message can be in the read queue at a time; if another message arrives, it is discarded and its message blocks freed.

## Processing message blocks

A process sends and receives characters on a stream using write(2) and read(2) system calls. When user data enters the stream head or external data enters the driver, the data is placed into message blocks for transmission on the stream. For upstream processing, these message blocks are transferred to the stream head, which extracts and copies the contents of the message blocks to user space. For downstream processing, the stream head copies data from user space to message blocks, which are sent to the driver.

## Message structures

Two message structures are contained in a message block:

- msgb, the message block
- datab, the data block

The msgb data structure links messages on a queue, links message blocks together, and manages read and write operations for the associated buffer (the data block). This structure contains pointers used to locate the data currently contained in the buffer.

The datab data structure points to the data block, which contains the message type, buffer limits, and control variables. This structure has pointers to the fixed beginning and end of the buffer.

# Queues

A **queue** is a data structure that is associated with a statically compiled module. Queues always come in pairs—one queue is for upstream (read) processing and the other is for downstream (write) processing. Figure 6-2 shows two modules, each of which consists of two queues.

**Figure 6-2**
Upstream and downstream queues

Each of the two queues are operated on independently from the other, so each can have different processing functions and data. As shown by the directional arrows in Figure 6-2, queues have direct access to the adjacent queue in the direction of message flow. A queue also has access to its mate's (upstream or downstream queue) messages and data.

A queue can contain or point to messages, processing procedures, or data. Messages are dynamically attached to the queue on a linked list as they pass through the module. A queue typically contains put and service routines (see "The Put Routine" and "The Service Routine" in this chapter), a message queue, and a private data area. The read queues in a module also contain the open and close procedures for the module. A developer may choose to provide private data if required to perform message processing (for example, state information and translation tables).

Three data structures form each queue:

- `queue_t` is the primary structure, which contains various modifiable values for the queue. Only the contents of `q_ptr` (pointer to a private data structure), `q_minpsz` (minimum packet size accepted by this queue), `q_hiwat` (message queue high-water mark), and `q_lowat` (message queue low-water mark) can be modified.
- `qinit` is a pointer to queue-processing procedures. A single common `qinit` structure pair is shared among all the queue pairs opened from the same `cdevsw` entry. All modules and drivers with the same `streamtab` (that is, fmodsw or `cdevsw` entry) point to the same upstream and downstream `qinit` structure or structures. This module is read-only.
- `module_info` contains identification and limit values. All modules and drivers with the same `streamtab` point to the same upstream and downstream `module_info` structure or structures. This module is read-only; however, the four limit values are copied to `queue_t`, where they can be modified.

## Driver flow control

Flow control is the Streams mechanism that regulates the flow of messages within a stream and the flow from user space into a stream.

To control downstream (write) flow, you can set flow control values (`mi_hiwat`) and (`mi_lowat`) in the downstream `module_info` structure. Streams then copies this information into the `q_hiwat` and `q_lowat` fields in the queue structure of the queue to set high-water and low-water marks. When a message is passed to the downstream put procedure, this procedure determines whether the device is busy. If so, it calls `putq` to enqueue the message. `putq` checks to see if the enqueued character count exceeds the high-water limit and halts message transmission until the count falls below the low-water mark (`q_lowat`).

Upstream (read) flow control is done with the `noenable` and `qenable` utilities. `noenable` disables the driver read service procedure. Messages are sent if the driver input interrupt routine determines that messages can be sent upstream. Otherwise, the message is enqueued until the queue becomes unblocked. `qenable` allows a module or driver to be scheduled.

An example of how to use these two routines would be a buffer module that calls `noenable` to inhibit its `service` procedure and its put procedure to enqueue messages until a certain byte count or time has been reached. Then the module could call `qenable` to gather messages in its message queue and forward them as a single, larger message.

## Utility routines

Streams provides a number of utility routines that you can use to write your Streams driver. The following list describes the function of and arguments to each of these routines. For a complete description of each routine, including calling sequence and parameters, see AT&T's *Streams Programmer Guide.*

| Utility | Function |
|---|---|
| `allocb` | Allocate a message block. The arguments to this routine are the minimum size of the data buffer and the priority of the allocation request. |
| `backq` | Get a pointer to the previous queue. The argument to this routine is a pointer to the current queue. |
| `canput` | Test for room in a queue. The argument to this routine is a pointer to the queue to be searched. |
| `copyb` | Copy a message block. The argument to this routine is a pointer to the message block to be copied. |
| `copymsg` | Copy a message. The argument to this routine is a pointer to the message block to be copied. |
| `dubp` | Duplicate a message block descriptor. The argument to this routine is a pointer to the message block descriptor to be duplicated. |
| `dupmsg` | Duplicate a message. The argument to this routine is a pointer to the message to be duplicated. |
| `flushq` | Flush a queue. The arguments to this routine are a pointer to the queue where message queue resides and a flag indicating what type of messages will be flushed. |

| | |
|---|---|
| `freeb` | Free a message block. The argument to this routine is a pointer to the message block descriptor to be freed. |
| `freemsg` | Free all message blocks in a message. The argument to this routine is a pointer to the message containing message blocks to be freed. |
| `getq` | Get a message from a queue. The argument to this routine is a pointer to the queue containing the message to be removed. |
| `linkb` | Concatenate two messages. The argument to this routine are pointers to the two messages to be concatenated. |
| `msgdsize` | Get the number of data bytes in a message. The argument to this routine is a pointer to the message containing data bytes to be returned. |
| `OTHERQ` | Get a pointer to the mate queue. The argument to this macro is a pointer to a queue (read or write) whose mate queue pointer is returned. |
| `putbq` | Return a message to the beginning of a queue. The arguments to this routine are pointers to a queue where the message will be returned and to the message itself. |
| `putctl` | Put a control message. The arguments to this routine are a pointer to a queue where the `put` procedure is located and the control message type. |
| `putctl1` | Put a control message with a 1-byte parameter. The arguments to this routine are a pointer to a queue where the `put` procedure is located, the message type, and a 1-byte parameter. |
| `putnext` | Put a message to the next queue. The arguments to this macro are a pointer to the calling queue and a pointer to the message to be passed. |
| `putq` | Put a message on a queue. The arguments to this routine are a pointer to the queue where the message queue is located and a pointer to the message to be put on the queue. |
| `qenable` | Enable a queue. The argument to this routine is a pointer to the queue to be enabled. |
| `qreply` | Send a message to a stream in the reverse direction. The arguments to this routine are a pointer to the originating queue and a pointer to the message to be sent. |
| `qsize` | Find the number of messages in the queue. The argument to this routine is a pointer to the queue where the messages are located. |
| `RD` | Get a pointer to the read queue. The argument to this routine is a pointer to the write queue in the same module. |

| | |
|---|---|
| `rmvb` | Remove a message block from a message. The argument to this routine are a pointer to the message block and a pointer to a message. |
| `splstr` | Set processor level. There are no arguments to this routine. |
| `unlinkb` | Remove a message block from the message head. The argument to this routine is a pointer to the first message block. |
| `WR` | Get a pointer to the write queue. The argument to this routine is the read queue pointer. |

## Streams device/module routines

The following routines are found in every Streams device driver or module.

### The open routine

The device open routine is called every time a process opens a device. This causes a kernel resident driver to be connected with a stream head to form a stream. A stream is created on the first `open(2)` system call made to a character special file corresponding to a Streams driver. A driver open routine has user context, so it can gain access to the u-dot and may call `sleep()`, although it must always return to the caller. In Streams open routines, all sleeps must be done with the `PCATCH` option (see `sleep`(kernel) in Appendix B). If the sleep returns, then the open routine should return failure.

### The close routine

The last `close(2)` system call dismantles the stream and closes the file. Dismantling consists of popping any modules on the stream, and closing the driver and the file. The close routine can delay before popping any modules to allow any messages on the module's write message queue to be drained by module processing. On return from the driver close routine, any message left on the driver's message queues are freed, and the `queue_t` and header structures are deallocated. Like the open routine the driver close routine has user context, so it can gain access to the u-dot and may call `sleep`, although it must always return to the caller.

The close routine closes a device. It is called when the last process that has the device open closes it. Note that this routine is called once, while the open routine is called many times.

*Note:* Streams frees only the messages contained on a message queue. The driver close routine must free any messages used internally by the driver.

## The put routine

A queue's put routine receives messages from the preceding queue. It provides the only entry point into one queue from a preceding queue. This routine first receives a message, does optional processing on it, then calls the `putq` utility. `putq` places the message on the tail of the message queue, schedules the queue for execution, then calls the service routine.

Put routines are generally required in pushable modules and there should be separate routines for upstream and downstream processing. Each queue must define a put routine in its `qinit` structure for passing messages between modules. A put routine must use the `putq` utility to enqueue a message on its own message queue. This is needed to maintain the fields of the `queue_t` structure consistently.

Put routines must never sleep because they have no user context.

## The service routine

A queue's optional service routine receives messages queued by the put routine. The main purpose of a service routine is delayed processing. It must be present for flow control.

The service routine gets the first message from the message queue with the getq utility, processes the message and passes it to the put procedure of the next queue with putnext. This processing continues in a first-in-first-out basis until the queue is empty or flow control blocks further processing, after which the service routine returns to the calling program. Service routines are optional. They have no user context, so they must never call the kernel `sleep` routine. A service routine must return to the caller after execution.

The service routine must use the Streams `getq` utility to remove a message from its message queue. The service routine should process all messages on its message queue unless the stream is blocked. To process a message, a service routine must do the following:

1. Remove the next message from the message queue using `getq`. If there is no message, return.
2. If the `canput` utility fails, this is not a priority message, and the message is to be put on the next queue, then go to step 3. Otherwise, go to step 4.
3. Replace the message using `putbq`, and exit the service procedure. Flow control will back-enable the service procedure. Back-enable is described later in this section.
4. Process the message as necessary and return to step 1.

Queues have both high-water and low-water marks. The high-water mark is the maximum number of messages that can be put on a queue (say *q1*). The low-water mark is the level at which a queue can begin to schedule new messages. After the high-water mark for *q1* is reached, new messages are put on another queue (*q2*) and the canput routine returns 0. This routine also sets a flag in *q1* so when its low-water mark is reached, *q2* will be scheduled for service. This process is known as a **back-enable**.

The put and service procedures give your driver rapid response along with queueing functions. The put procedure allows rapid response to certain data and events such as software echoing of input characters, because it is granted a higher priority than schedules service routines. Queueing defers processing of the service procedure until all queues are processed. Service routines allow processing time to be more evenly spread between multiple streams.

## Streams scheduling

When a message is placed on an empty queue, it is *scheduled*. This means that its service routine will be called in the near future after all interrupts are serviced and the processor is running at processor level `spl0`. Service routines have no process context. Other ways to schedule a queue are by means of the `qenable` routine and by back-enabling from flow control.

## Cloned devices

A cloned device is a Streams device that returns an unused minor device number when initially opened, rather than requiring the minor device number to be specified in the open(2) call. Cloned devices can be useful when a user process wants to connect a new stream to a driver, regardless of which minor device is opened. To help your driver open a cloned device, Streams provides the clone open facility. The clone driver (see `clone(7)`) is a system-dependent Streams pseudo-device driver.

When an open(2) system call is made to a cloned device's Streams file, open causes a new stream to be opened to the clone driver and the open procedure in the clone to be called.

A cloned device has a major number corresponding to the clone device driver and minor number corresponding to the major number of the target driver.

# Chapter 7

# Streams Terminal Drivers

This chapter describes how to write a Streams-based terminal driver. In particular the chapter describes how to use a group of A/UX kernel routines called the `ttx` library. The purpose of these routines is to make Streams terminal drivers work like traditional character drivers. The main difference between traditional character drivers and Streams drivers is that Streams drivers deals with messages and queues, rather than cblocks and clists.

If you wish, you can write your own Streams-based terminal driver. The advantage of using the `ttx` library package is that it provides almost all the Streams interfacing code, so it makes writing a Streams terminal driver that much easier.

*Note:* The `ttx` library is not a generic part of a Streams driver. If you wish to write a driver that is portable to other systems, you must not use this subroutine library. For general information about writing standard Streams drivers, see Chapter 6.

At the end of this chapter you'll find a skeleton Streams driver that you might want to use as a guide for writing your own driver.

## Streams line discipline

Streams is a mechanism that provides a way of controlling how information is processed on its way to and from devices. For TTY-style devices (such as terminals), this controlling mechanism is normally done using the Streams module **line,** which is actually a line discipline. The line processes characters as they are sent to and from a terminal. It provides functions such as:

- echoing
- erase and kill processing
- flow control
- `ioctl`(2) processing (see `termio`(7))
- character editing (for example, turning carriage returns into line feeds)

In traditional character drivers, both the driver and the line discipline perform these functions. In Streams terminal drivers, however, the driver is specifically responsible for output flow control (recognizing `XON/XOFF`) and ioctls from `termio`(7) that directly affect the device (in particular, the parts that control things like baud rate, parity, number of stop bits, and character size).

The line discipline does the rest. The two parts differ because the driver must be able to operate without the line discipline being present on the Stream. This structure allows greater efficiency in operations that don't require the line discipline. A Macintosh II device without a line discipline module pushed onto it is said to be operating in *raw* or *uncooked* mode.

Communicating with the line discipline is done by passing messages back and forth along the queue. Because the code to do this is the same for all character devices, the `ttx` library has been written to make it easier to write a Streams driver. Thus, you don't have to know about Streams in order to write a Streams terminal driver—you just have to know how to use a basic skeleton driver.

## Data structures

As mentioned in Chapter 6, the `streamtab`, `qinit`, and `module_info` data structures provide the driver interface to the operating system. They are set up once for each driver and reference each other, as well as the routines that perform various Streams functions. Remember that the following structures must reference the Streams put and service routines:

| | |
|---|---|
| `ttx_rsrvc` | Streams read service routine |
| `ttx_wputp` | Streams write put routine |
| `ttx_wsrvc` | Streams write service routine |

The reference to the Streams read put routine should be `NULL`. You must also add the addresses of your driver's open and close routines. The `streamtab` data structure must be declared `extern` because it is referenced externally and all the data structures are accessible from it. The other data structures should be declared `static`.

The open and close routines are the only ones that are ever called in process context. This means that they are the only ones that can reference the u-dot, copy data to and from processes, or call `sleep`. Because these routines are called from kernel routines that allocate dynamic data structures, they must always sleep with the `PCATCH` signal set. With `PCATCH` set, a signal is not delivered to wake the sleeping process until after the open or close is complete and the dynamic data structures (such as the queues and the stream head) are disposed of.

Each Streams `tty` structure must have a data structure allocated for it of type `struct ttx`. This data structure is normally called the **`ttx` structure**. It is referenced by the device's stream queue (via the `q_ptr` field) and contains most of the context that is needed for operating a terminal-style device. The `ttx` structure contains the following fields:

| | |
|---|---|
| `t_q` | Pointer to the read queue attached to this device |
| `t_rm` | Pointer to the current input buffer |
| `t_xm` | Pointer to the current output buffer |
| `t_proc` | Address of the device's command process routine (required) |
| `t_ioctl` | Address of the device's ioctl routine (optional) |

| | |
|---|---|
| `t_dev` | Device ID (for user only); normally the minor number |
| `t_addr` | Device's address (for user only) |
| `t_count` | Number of bytes remaining in the input buffer |
| `t_size` | Size of an empty input buffer (set from the size parameter to `ttxinit`) |
| `t_iflag` | Input-processing modes from `TCSETA` (see `termio`(7)) |
| `t_oflag` | Output-processing modes from `TCSETA` |
| `t_cflag` | Device modes from `TCSETA` |
| `t_lflag` | Line discipline modes from `TCSETA` |
| `t_state` | Current device state. The defined flags for `t_state` are as follows: |

| | |
|---|---|
| `BUSY` | *Device is currently transmitting. |
| `TTSTOP` | *Output is stopped. |
| `TTXOFF` | *Send an `XOFF` as soon as possible. |
| `TTXON` | *Send an `XON` as soon as possible. |
| `TBLOCK` | *Input is blocked (via an `XOFF`). |
| `TIMEOUT` | Device is sending a line break. |
| `XMT_DELAY` | Device has stopped transmitting because of delay (usually after a newline or other cursor motion character). |
| `OASLP` | `ttx` library is waiting for output to drain so it can complete a close. |
| `RCV_TIME` | System is out of buffers for receiving, so it's trying to obtain more. |
| `WOPEN` | *One or more processes are waiting for carrier before they open the device. |
| `ISOPEN` | *At least one process has the device open. |
| `CARR_ON` | *Carrier line is turned on. |

*Note:* Only the values marked with as asterisk (*) can be changed; the others are used internally by the `ttx` library.

## The Streams terminal driver routines

The Streams terminal driver routines that you must write are listed below. In all cases, *prefix* is the device prefix used your driver.

- *prefix*`init` initializes the device. The kernel calls it once before interrupts are turned on.
- *prefix*`open` opens a device. It is called every time the device is opened.
- *prefix*`close` closes a device. It is called when the last process that has a device open closes it.
- *prefix*`ioctl` performs special functions. It is called whenever an unknown ioctl message from a process is received at the driver.
- *prefix*`param` sets up hardware. It is called internally to set up device parameters such as baud rate.
- *prefix*`int` handles interrupts. It is called as a result of a device interrupt.
- *prefix*`proc` performs command processing. It is called internally and also by the `ttx` library whenever it wants something to be done.

*Note:* All of these routines except *prefix*`int` and *prefix*`init` are normally declared static. This is because they either are internal routines that are never called externally or are referenced by a data structure such as the stream description (`streamtab`) or the `ttx` structure.

## The open routine

The streams device open routine is called every time a process opens a device. The streams device open routine has the following format:

```
static int
prefixopen(q, dev, flag, sflag, err)
queue_t *q;
dev_t dev;
int flag;
int sflag;
int *err;
```

where

- *q* is a pointer to the read queue for the device end of the stream.
- *dev* is the device number of the device.
- *flag* is the normal device open flags passed to the open routine.
- *sflag* is the Streams flag. Possible values for *sflag* are as follows:

| | |
|---|---|
| `MODOPEN` | A module is being opened (pushed). |
| `DEVOPEN` | A normal device is being opened. |

CLONEOPEN A cloned device is being opened (if successful, the device's minor number is returned by the open routine).

- *err* is a pointer to a location where any errors are stored if the open fails.

*Note:* The open routine arguments are slightly different from normal Streams open routines. The vnode kernel requires an open routine to return errors to the caller rather than placing place them in `u.u_error`, as is done in other systems. Remember this when you port drivers to other Streams implementations. For more information on vnode kernel changes, see Appendix E of this manual.

## The close routine

The streams close routine closes a device. It is called on the last close of a device. Note that this routine is called once, while the open routine is called many times. The streams close routine is called as follows:

```
static int
```

*prefix*`close(`*q*, *flag*`)`

```
queue_t *q;
int flag;
```

where

- *q* is the read queue of the stream being dismantled.
- *flag* is the flag passed to the open routine.

## The initialization routine

The initialization (or *init*) routine puts the device into a known state. The system invokes these routines once during system initialization. By using autoconfiguration, you have a choice of where such initialization occurs (see Chapter 12 for details). Normally, an initialization routine is called before interrupts are turned on. Here is the format of the initialization routine:

`int` *prefix*`init()`

## The parameter routine

A device's parameter (or *param*) routine is called internally to set up registers in the device using values stored in the `ttx` structure passed to it. The format of the param routine is as follows:

```
static int
prefixparam(tp)
register struct ttx *tp;
```

where *tp* is the device's `ttx` structure.

## The ioctl routine

The streams ioctl routine performs message handling. This routine referenced by the `t_ioctl` field in the `ttx` structure, and is called when an ioctl message is received at the device end of the stream that can't be handled by the `ttx` library. The streams ioctl routine has the following format:

```
static int
prefixioctl(tp, iocbp, args)
struct ttx *tp;
struct iocblk *iocbp;
mblk_t *args;
```

where

- *tp* is the device's ttx structure.
- *iocbp* is a pointer to the ioctl message's control block (the first block in the message).
- *args* is a pointer to the entire ioctl message.

If the ioctl routine returns a nonzero result, the routine failed and the message is sent back to the stream head as an error (an error return may be placed in the I/O control block, if desired). The ioctl routine is optional, but if not present, all unknown messages return an error value. If parameter(s) were sent to the ioctl call they appear in the second (and subsequent) message block of the message referenced by *args*. You may need to allocate message blocks to hold data returned to a user program.

## The command process routine

The command process (or *proc*) routine processes commands requested by the system and other parts of the driver. The `t_proc` field in the `ttx` structure contains the address of the command process routine. The `ttx` library calls this routine (and a driver calls it internally) whenever a driver must perform an action. The command process routine is required. The command process routine has the following format:

```
static int
```

*prefix*`proc(`*tp, cmd*`)`

```
register struct ttx *tp;
int cmd;
```

where

- *tp* is a pointer to the ttx structure that identifies the device.
- *cmd* is a command requesting an action (or notifying the driver of a change).

This routine should always disable device interrupts upon entry (as it can be called from a device interrupt routine), and return them to their previous state (using `splx`) upon exit. Most of what a command process routine does is the same from device to device, because the device-dependent parts are usually simple things like transmitting a character or starting a line break. The reason for repeating much of this code is to support drivers for intelligent devices (for example, ones that can do DMA or that have large internal buffers). The following commands are passed when calling a device's command process routine:

| | |
|---|---|
| `T_BREAK` | Start transmission of a line break. When this happens, `ttx_break` (see below) should be called so that a `T_TIME` call will be made later. |
| `T_TIME` | Complete transmission of a line break and resume normal output. |
| `T_WFLUSH` | Discard any characters queued for output (some devices have internal queues that should also be flushed). |
| `T_RFLUSH` | Flush any characters waiting to be input. |
| `T_RESUME` | Restart suspended output (usually by an `XOFF` or a user ioctl request). |
| `T_SUSPEND` | Suspend output until a `RESUME` occurs. On devices with large internal buffers, some special action may be required to stop output. |
| `T_OUTPUT` | Start output if possible. This is usually done when a device transmitter interrupt occurs or a data message arrives at the device's queue. |
| `T_BLOCK` | Block input (by sending an `XOFF` to the remote end). |
| `T_UNBLOCK` | Unblock input (by sending an `XON` to the remote end). |
| `T_PARM` | Call driver's parameter routine because the device's parameters (in the `ttx` structure) have changed. |
| `T_INPUT` | A new input buffer is available. For simple devices, this is ignored. For devices that do DMA directly into device buffers, `T_INPUT` is used to tell the device about a new buffer. |

These commands are the same as those used in traditional character drivers. This is done so it's easier to transport old drivers to the new Streams style of driver writing.

## The ttx library support routines

Streams terminal drivers can call the `ttx` library support routines described in this section.

### ttxinit

The `ttxinit` routine has three main purposes. First, it initializes the `ttx` structure passed to it. Second, it associates the stream's queue and the device's `ttx` structure (by having them point to each other), so that the device is associated with the stream. Finally, it allocates a receive buffer for the driver using the size passed. If the size is zero, then no buffer is allocated. You might not want to allocate buffers for devices (such as printers) that can't receive characters, or for smart devices that may wish to manage their own receive buffering.

A driver's open routine calls `ttxinit` when a device is first opened. Before calling it, the `t_ioctl` field in the `ttx` structure field must point to your driver's command process routine. The `ttxinit` routine has the following format:

```
int ttxinit(q, tp, sz)

queue_t *q;

struct ttx *tp;

int sz;
```

where

- *q* is the queue pointer passed to the open routine.
- *tp* is the `ttx` structure to be associated with the device.
- *sz* is the size of the input buffer for this device.

### ttx_put

Receive interrupt routines that have placed characters in the receive buffer (pointed to by the `ttx` field `t_m`) call the `ttx_put` routine to pass the message down the Stream and, if possible, to allocate a new buffer. The `ttx_put` routine returns a nonzero value if it can't allocate a buffer. Routines in the `ttx` library will continue to try to allocate a buffer until it succeeds. The `ttx_put` routine has the following format:

```
int ttx_put(tp)

struct ttx *tp;
```

where *tp* is the ttx structure to be associated with the device.

## ttx_sighup

The `ttx_sighup` routine notifies processes that have a device open that the driver has detected a hangup. The routine also flushes any queued input and output. The `ttx_sighup` routine has the following format:

```
int ttx_sighup(tp)
```

where *tp* is the `ttx` structure to be associated with the device.

## ttx_break

A driver calls the `ttx_break` routine to handle break processing. The routine marks the `ttx` structure so that no output can occur during the break and starts a timeout to wake up the driver using `T_TIME` to stop the break. The `ttx_break` routine has the following format:

```
int ttx_break(tp)

struct ttx *tp;
```

where *tp* is the `ttx` structure to be associated with the device.

## ttx_close

The `ttx_close` routine performs close operations as part of a device close. The `ttx_close` routine waits for output to drain, flushes input and output, discards buffers, and then breaks the connection between the queue and the device. The `ttx_close` routine has the following format:

```
int ttx_close(tp)

struct ttx *tp;
```

where *tp* is the `ttx` structure to be associated with the device.

# A Skeleton Streams driver

You can use the following example as a template for Streams terminal drivers. The comments marked with the string `DEV` should be replaced by device-dependent code that performs the actions described. Of course, not all devices are completely straightforward, so it may be necessary to make additional changes to the driver.

```
#include "sys/Stream.h"

#include "sys/tty.h"
```

```
#include "sys/ttx.h"
#define NDEVICES 4          /* the number of devices supported */
static struct ttx DEV_tty[NDEVICES];       /* the per-device "ttx" structures */
extern int nulldev();                      /* externally defined routines */
extern int qenable();
/*
 *     Locally declared routines that need to be
 *             declared before use
 */
static int DEVopen();
static int DEVclose();
static int DEVproc();
static int DEVioctl();
/*
 *     The following four data structures collectively describe the
 *     interface to the Streams system. Note that they reference these
 *     ttx routines:
 *
 *             ttx_rsrvc
 *             ttx_wputp
 *             ttx_wsrvc
 *
 *     which intercept messages sent to the device and convert them into
 *     calls to the device's command process routine below.
 *
 *     These routines must reference your driver's open and close
 *     routines.
 *
 *     The structure DEVinfo is the primary interface with the rest of
 *     the kernel (it references the other three). It, the interrupt
```

```
 *      routine DEVint, and the initialization routine DEVinit are the
 *      only data structures that need to be declared and referenced
 *      externally to this driver.
 *
 */
static struct module_info DEV_info = {5321, "DEV", 0, 256, 256, 256};
static struct qinit DEV_rq = {NULL, ttx_rsrvc, DEVopen, DEVclose,
                              nulldev, &DEV_info, NULL};
static struct qinit DEV_wq = {ttx_wputp, ttx_wsrvc, DEVopen, DEVclose,
                              nulldev, &DEV_info, NULL};
struct Streamtab DEVinfo = {&DEV_rq, &DEV_wq, NULL, NULL};
/*
 *      The initialization routine is called with interrupts disabled. Its
 *      purpose is simply to put the device into a known, stable state.
 */
DEVinit()
{
        register int count;
        for (count = 0; count < NDEVICES; count++) {
                /* DEV: Initialize the device referenced by count */
        }
}
/*
 *      The device's open routine is called whenever the device is opened.
 *      When the device is first opened it must be prepared for use (for
 *      example, it should be set to its initial baud rate and its
 *      interrupts should be turned on). Also when it is first turned on
 *      its corresponding ttx structure is initialized and ttxinit() is
 *      called to complete this initialization and to allocate receive
 *      buffers for it.
```

```
 *
 *      Since Streams open and close routines are called from process
 *      context (i.e., in the context of the process that is doing the
 *      open or close) they can sleep(). But because they are Streams open
 *      routines they must sleep with the PCATCH flag set. This is because
 *      the stream open causes the Stream data structures to be built; if
 *      an open fails because of a signal, the open routine must catch the
 *      signal and return OPENFAIL.
 *
 *      This example shows a driver that only supports modem control.
 *      Note that only one process actually ever sleeps waiting for
 *      carrier presence. All others sleep waiting for that process to
 *      finish before proceeding. This is because if the process that did
 *      the initial open (the one that called ttxinit()) fails, it MUST
 *      call ttx_close in order to free the buffer that was allocated for
 *      input.
 *
 */
static
DEVopen(q, dev, flag, sflag, err)
queue_t *q;
dev_t dev;
int *err;
{
        register struct ttx *tp;
        struct device *device;
        dev = minor(dev);
        if (dev >= NDEVICES) {                  /* Check the device ID for */
                *err = ENXIO;                   /* validity */
                return;
```

```
}
tp = &DEV_tty[dev];
while (tp->t_state & WOPEN) {                   /* Sleep until other opens */
        if (sleep((caddr_t)&tp->t_q, TTOPRI|PCATCH)) /* complete */
                return(OPENFAIL);
}
if ((tp->t_state&(ISOPEN|WOPEN)) == 0) { /* If this is the first open: */
        tp->t_proc = DEVproc;                   /* initialize the ttx */
        tp->t_ioctl = DEVioctl;         /*      structure */
        ttxinit(q, tp, 4);              /* allocate a 4-byte rcv */
                                        /*      buffer*/
        /* DEV: Put the devices chip address in addr */
        tp->t_addr = (caddr_t) (addr);
        tp->t_dev = dev;
        /* DEV: Assert the DTR line */
        DEVparam(tp);                   /*      Set up the device */
        /* DEV: Put the state of the DCD line in dcd */
        if (dcd)                        /*      if carrier mark it */
                tp->t_state |= CARR_ON;
}
if (!(flag & FNDELAY)) {                /* Sleep until carrier*/
                                                /* present*/
        while ((tp->t_state & CARR_ON) == 0) {
                tp->t_state |= WOPEN;
                if (sleep((caddr_t)&tp->t_q, TTOPRI|PCATCH)) {
                        if (!(tp->t_state&ISOPEN))   /* If interrupted */
                                ttx_close(tp);              /* a signal exit */
                        tp->t_state &= ~WOPEN;              /* gracefully */
                        wakeup((caddr_t)&tp->t_q);
                        return(OPENFAIL);
```

```
                }
            }
        }
        tp->t_state &= ~WOPEN;                  /* Mark the device open */
        tp->t_state |= ISOPEN;
        return(1);                              /* Return success */
}
/*
 *      The close routine's main purpose is to call ttx_close to wait for
 *      output to drain and then recover the input buffers. After this,
 *      DTR is removed (if required).
 *
 *      It is only called when the last process has the device open closes
 *      it, i.e., just before the system dismantles the Stream data
 *      structures.
 *
 *      Note:  Close routines are also called from process context,
 *             so they can sleep (again they must use PCATCH). However this
 *             is unusual. Close routines always succeed and so don't
 *             return status.
 */
/* ARGSUSED */
static
DEVclose(q, flag)
queue_t *q;
int flag;
{
        register struct ttx *tp;
        int s;
        tp = (struct ttx *)q->q_ptr;
```

```
        ttx_close(tp);
        if (tp->t_cflag & HUPCL) {
                /* DEV: Hang up the device (remove DTR) */
        }
}
/*
 *      Interrupt service routines depend on the type of device being
 *      used. This one assumes that there are three basic types of events
 *      signalled by the device: transmit, receive and DCD change.
 */
DEVint(ap)
struct args *ap;
{
        int type, dev;
        /* DEV: Figure out from the a_dev field in ap and the device which
                device caused the interrupt and put it in dev */
        /* DEV: Figure out the type of interrupt (receive/transmit/special
                condition) and put a code in type */
        switch(type) {
        case 0:
                DEVrintr(dev);
                break;
        case 1:
                DEVtintr(dev);
                break;
        case 2:
                DEVsintr(dev);
                break;
        }
}
```

```
/*
 *      Receive interrupt routines basically read a character and status,
 *      process flow control, process errors and then pass the character
 *      back to the queue by putting it into a message buffer and calling
 *      ttx_put(). If no buffers are available, characters are discarded.
 */
static
DEVrintr(dev)
int dev;
{
        register mblk_t *m;
        register struct ttx *tp;
        register int c;
        int s, lcnt, flg;
        char ctmp;
        char lbuf[3];
        sysinfo.rcvint++;
        tp = &DEV_tty[dev];
        /* DEV: Read the device status register and put it in s */
        /* DEV: Read the received character register and put it in c */
        /*
         *      If output software flow control is enabled and the character
         *      is an XON/XOFF character then call the command process
         *      routine to stop output of characters. Note: this happens
         *      even if the input character was found to be in error.
         */
        if (tp->t_iflag&IXON) {
                ctmp = c & 0x7f;
                if (tp->t_state&TTSTOP) {
                        if (ctmp == CSTART || tp->t_iflag&IXANY)
```

```
                        DEVproc(tp, T_RESUME);
                } else {
                        if (ctmp == CSTOP)
                                DEVproc(tp, T_SUSPEND);
                }
                if (ctmp == CSTART || ctmp == CSTOP)
                        return;
        }
        /*
         *      If no buffers are available, throw the character away
         */
        if ((m = tp->t_rm) == NULL)
                return;
        /*
         *      Check for errors
         */
        lcnt = 1;
        flg = tp->t_iflag;
        /*
         *      Decode the device-dependent errors
         */
        if (s & (C_PERR|FRERR|ROVRN|RA_B)) {            /* These bits are device
                                                        /*      dependent */
                if (s & C_PERR)
                        c |= PERROR;
                if (s & FRERR)
                        c |= FRERROR;
                if (s & ROVRN)
                        c |= OVERRUN;
                if (s & RA_B)
```

```
                c = FRERROR;                    /* Clear c for break */
}
/*
 *      Now do device-independent error processing
 */
if (c&(FRERROR|PERROR|OVERRUN)) {
        if ((c&0xff) == 0) {                    /* A break was detected */
                if (flg&IGNBRK)
                        return;
                if (flg&BRKINT) {               /* Send a message to the */
                                                /*      line discipline */
                        putctl1(tp->t_q->q_next, M_CTL, L_BREAK);
                        return;
                }
        } else {
                if (flg&IGNPAR)                 /* Ignore characters in error */
                        return;
        }
        if (flg&PARMRK) {                       /* Pass back marked characters */
                                                /*      in error */
                lbuf[2] = 0xff;
                lbuf[1] = 0;
                lcnt = 3;
        } else
                c = 0;
} else {                                        /* the normal case */
        if (flg&ISTRIP)
                c &= 0x7f;
        else {
                c &= 0xff;
```

```
                if (c == 0xff && flg&PARMRK) {
                        lbuf[1] = 0xff;
                        lcnt = 2;
                }
        }
}
/*
 *      Copy the characters out from the temporary buffer to
 *      the Streams buffer, then call ttx_put to send
 *      it to the line discipline when we are finished or
 *      the buffer is full.
 */
if (lcnt != 1) {
        lbuf[0] = c;
        while (lcnt) {
                *m->b_wptr++ = lbuf[--lcnt];
                if (--tp->t_count == 0) {
                        if (ttx_put(tp))
                                return;
                        if ((m = tp->t_rm) == NULL)
                                return;
                }
        }
} else {
        *m->b_wptr++ = c;
        tp->t_count--;
}
if (m && m->b_wptr != m->b_rptr)
        (void) ttx_put(tp);
}
```

```
/*
 *      The transmit interrupt routine clears the BUSY flag and then calls
 *      the command process routine to send the next character.
 */
static
DEVtintr(dev)
int dev;
{
        register struct ttx *tp;
        sysinfo.xmtint++;
        tp = &DEV_tty[dev];
        tp->t_state &= ~BUSY;
        DEVproc(tp, T_OUTPUT);
}
/*
 *      The external status change interrupt routine does two things:
 *
 *              signals the presence of carrier (DCD) to any processes
 *              waiting for opens (see the open routine above for
 *              the other half of this handshake)
 *
 *              detects the loss of carrier and calls ttx_sighup() to
 *              send this signal down the stream to waiting processes
 */
static
DEVsintr(dev)
int dev;
{
        register struct ttx *tp;
        int dcd;
```

```
        sysinfo.mdmint++;
        tp = &DEV_tty[dev];
        /* DEV: Assign the current value of the DCD line to dcd */
        if (dcd) {
                if ((tp->t_state & CARR_ON) == 0) {
                        tp->t_state |= CARR_ON;
                        if (tp->t_state & WOPEN)
                                wakeup((caddr_t)&tp->t_q);
                }
        } else {
                if (tp->t_state & CARR_ON) {
                        tp->t_state &= ~CARR_ON;
                        ttx_sighup(tp);
                }
        }
}
/*
 *      This routine is for device-specific ioctls. The iocbp pointer
 *      refers to the stream message header for the ioctl (see
 *      <sys/Stream.h> for more info). This contains the ioctl type and
 *      who made it. If an ioctl succeeds you must return 0 from this.
 *      Otherwise, you should return 1 for errors (and optionally fill in
 *      the error field in the ioctl message). This example shows no
 *      actual ioctls. Often devices might use this to change device-
 *      dependent functions such as turning on or off modem control
 *      or flow control.
 *
 *      The ttx library intercepts most of the ioctls from termio(7) and
 *      converts into calls to the drivers command process and parameter
 *      routines.
```

```
 */
static
DEVioctl(tp, iocbp, args)
struct ttx *tp;
struct iocblk *iocbp;
mblk_t *args;
{
        return(1);
}
/*
 *      The device's parameter routine is called whenever an ioctl is made
 *      that may have changed the device specific functions (such as baud
 *      rete, parity etc.). Care should be taken on some chips that
 *      require output to complete before such changes so that characters
 *      are sent without errors.
 */
static
DEVparam(tp)
register struct ttx *tp;
{
        register int s;
        register flag;
        flag = tp->t_cflag;
        if ((flag&CBAUD) == 0) {
                /* DEV: Do device hangup (remove DTR) */
                return;
        }
        s = splstr();
        /*
         *      DEV: Set up the following:
```

```
         *
         *                    baud rate
         *                    number of bits per character
         *                    parity (on/off/odd/even)
         *                    number of stop bits
         *
         *      from the ttx field t_cflag (see tty.h for defines)
         /
        splx(s);
}
/*
 *      The command process routine is the place where device-dependent
 *      actions are requested by the system and other parts of the driver.
 *      Each call has a command that describes what is being requested.
 */
static
DEVproc(tp, cmd)
register struct ttx *tp;
{
        register int s, c, x;
        register mblk_t *m, *ml;
        s = splstr();
        switch (cmd) {
        case T_TIME:
                /*
                 * TIME              Stop an output break condition and continue
                 *             normal output.
                 */
                /* DEV: Clear break condition on device */
                goto start;
```

```
case T_WFLUSH:
        /*
         * WFLUSH       Request that any pending output should be
         *              discarded
         */
        /* DEV: Flush any pending output characters from device */
        if (tp->t_xm) {
                freemsg(tp->t_xm);
                tp->t_xm = NULL;
        }
        /* fall through */
case T_RESUME:
        /*
         * RESUME       Restart output after it being SUSPENDed
         */
        tp->t_state &= ~TTSTOP;
        goto start;
case T_OUTPUT:
        /*
         * OUTPUT       Send a character (if the device is not already
         *              doing something). Also send XON/XOFF
         *              characters when required.
         */
start:
        if (tp->t_state&(TIMEOUT|TTSTOP|BUSY|XMT_DELAY) || !tp->t_q)
                break;
        if (tp->t_state & TTXON) {
                c = CSTART;
                /* DEV: Transmit the character in c */
                tp->t_state |= BUSY;
```

```
                tp->t_state &= ~TTXON;
                break;
        } else if (tp->t_state & TTXOFF) {
                c = CSTOP;
                /* DEV: Transmit the character in c */
                tp->t_state |= BUSY;
                tp->t_state &= ~TTXOFF;
                break;
        }
        m = tp->t_xm;                           /* If nothing to transmit, */
        if (m == NULL) {                        /* then wake up the */
                qenable(WR(tp->t_q));                   /* Streams output handler */
                break;
        }
        c = *m->b_rptr++;                       /* Get a character */
        /* DEV: transmit the character in c */
        tp->t_state |= BUSY;
        while (m->b_rptr >= m->b_wptr) {/* Remove empty messages */
                m1 = unlinkb(m);                /*      from the output buffer*/
                freeb(m);
                tp->t_xm = m = m1;
                if (m == NULL) {        /* If nothing is left then break */
                        break;
                }
        }
        break;

case T_SUSPEND:
        /*
         *      SUSPEND         Stop output until it is RESUMEd
```

```
         */
        tp->t_state |= TTSTOP;
        break;
case T_BLOCK:
        /*
         *      BLOCK   Send an XOFF to signal the other end not to
         *              send any more
         */
        tp->t_state |= (TBLOCK|TTXOFF);
        tp->t_state &= ~TTXON;
        goto start;
case T_RFLUSH:
        /*
         *      RFLUSH          Discard any received input
         */
        if (m = tp->t_rm) {
                tp->t_count = tp->t_size;
                m->b_wptr = m->b_rptr;
        }
        /* DEV: Flush any received characters from the device */
        if (!(tp->t_state&TBLOCK))
                break;
case T_UNBLOCK:
        /*
         *      UNBLOCK                 Send an XON to signal the other end to
         *                      resume its transmission
         */
        tp->t_state &= ~(TTXOFF|TBLOCK);
        tp->t_state |= TTXON;
        goto start;
```

```
        case T_PARM:
                /*
                 *      PARM    Call the device's parameter routine to reflect
                 *              changes in the device's attributes.
                 */
                DEVparam(tp);
                break;

        case T_BREAK:
                /*
                 *      BREAK   Start transmission of a line break
                 */
                /* DEV: Start a break condition on the device */
                ttx_break(tp);
                break;
        }
        splx(s);
}
```

# Chapter 8

# Network Drivers

B-NET network facilities provide a uniform user interface to networking within the A/UX operating system. If you're implementing new communication protocols and network services, B-NET's network communications structure promotes code sharing and minimizes implementation effort. A major goal of the system is to provide a framework that makes it easier to support new protocols and hardware.

For a description of the data structures, utility routines, and internal layers of the B-NET network system, see "Networking Implementation Notes" listed in this manual's bibliography.

To illustrate how you could write a network driver, the rest of this chapter provides a sample network driver for Ethernet Version 1.0 and 2.0. The include file, `if_xx.h`, is listed first, followed by the sample driver, `if_xx.c`.

## Include file

```
#define        NXX             6
#include       <sys/via6522.h>
#define        PHYS            0xf0000000
#define        XXMEMBASE(unit)       ((unsigned)PHYS+((SLOT_LO + (unit)) << 24))
/*
 * Ethernet software status per interface.
 *
 * Each interface is referenced by a network interface structure,
 * xx_if, which the routing code uses to locate the interface. This
 * structure contains hardware dependent addresses and status, the
 * interface address and error counts for the interface.
 */
struct xx {
       struct arpcom xx_ac;                /* common ethernet structures */
#define        xx_if           xx_ac.ac_if    /* network-visible interface */
#define        xx_enaddr       xx_ac.ac_enaddr              /* hardware Ethernet address */
       short           xx_oactive;    /* output active flag */
       int             xx_flags;      /* flag bits */
       /* hardware-dependent variables go here */
```

```
};
/*
 *      bits in xx_flags
 */
#define         XX_TIMEOUTPENDING     1                    /* transmitter watchdog timer
running */
```

## Sample driver

```
#include        <sys/types.h>
#include        <sys/reg.h>
#include        <sys/mbuf.h>
#include        <sys/socket.h>
#include        <sys/ioctl.h>
#include        <sys/var.h>
#include        <sys/errno.h>
#include        <net/if.h>
#include        <net/route.h>
#include        <net/netisr.h>
#include        <netinet/in.h>
#include        <netinet/in_systm.h>
#include        <netinet/ip.h>
#include        <netinet/ip_var.h>
#include        <netinet/if_ether.h>
#include        <vaxuba/ubavar.h>
#include        "if_xx.h"
extern int xxcnt;
extern int xxaddr[];
int     xx_probe(), xx_init(), xx_attach(), xx_output(), xx_ioctl(),
        xxint(), xx_rint(), xx_tint(), xx_timeout();
struct mbuf *xx_get();
```

```
int xx_trans[16];
struct uba_device *xxinfo[NXX];
struct uba_driver xxdriver = {
        xx_probe, xx_attach, (u_short *) 0, xxinfo
};
static          struct          xx xx[NXX];
extern struct ifnet loif;
/*
 *      Called from the network initialization code, this function is
 *      responsible for confirming the existence of the device described
 *      in ui. In the context of autoconfiguration, you need only check
 *      that the device's unit number (ui->ui_unit) is reasonable.
 *      Take this opportunity to call xx_map(), which sets up the mapping
 *      between unit number, slot number, and the interface's board RAM.
 *
 *      Return value:   1: interface exists
 *                      0: interface does not exist
 */
static
xx_probe(ui)
 struct uba_device *ui;
{
        struct xx *xxp = &xx[ui->ui_unit];
        if (ui->ui_unit < xxcnt) {
                xx_map(ui);
                return (1);
        }
        else
                return (0);
}
```

```
/*
 *      Record the correspondence between unit number, slot number, and
 *      board RAM. On some systems, this function might arrange to map
 *      in the interface's RAM at a well-known address.
 *
 *      Return value: none.
 */
static
xx_map(ui)
        struct uba_device *ui;
{
        struct xx *xxp = &xx[ui->ui_unit];
        int ind;
        ind = xxaddr[ui->ui_unit] - SLOT_LO;
        xx_trans[ind] = ui->ui_unit;
        /* Set up device-specific pointers */
}
/*
 *      If the interface's probe routine returns 1 (indicating that the
 *      interface exists) the network initialization code will then call
 *      the interface's attach routine. The conventional purpose of this
 *      function is to initialize the fields in the ifnet structure (i.e.,
 *      the unit number interface name, maximum transmission unit, address
 *      family or families supported, and the device's initialization, I/O
 *      control and output routines) and call if_attach() to add itself to
 *      the system's list of known interfaces. Refer to section 5.3 of
 *      the Networking Implementation Notes for details on the ifnet
 *      structure.
 *
 *      Return value: none.
```

```
 */
static
xx_attach(ui)
        struct uba_device *ui;
{
        struct ifnet *ifp = &xx[ui->ui_unit].xx_if;
        struct sockaddr_in *sin;
        ifp->if_unit = ui->ui_unit;
        ifp->if_name = "xx";
        ifp->if_mtu = ETHERMTU;
        sin = (struct sockaddr_in *) &ifp->if_addr;
        sin->sin_family = AF_INET;
        ifp->if_init = xx_init;
        ifp->if_ioctl = xx_ioctl;
        ifp->if_output = xx_output;
        if_attach(ifp);
}
/*
 *      When the networking subsystem is ready to process packets or when
 *      the driver must reinitialize an interface, this function will be
 *      called. Nothing should be done until its address is known. It
 *      should then:
 *              reset the hardware to begin receiving packets
 *
 *              set the if_flags fields to indicate that it is up
 *              and has resources allocated
 *
 *              start output if there are packets on the send queue
 *
 *              call if_rtinit() to indicate the interface is up
```

```
 *              and may have packets routed through it
 *
 *              call arpwhohas() to announce its Ethernet and Internet
 *              addresses to the world
 *
 *      Return value:  none.
 */
static
xx_init(unit)
 int    unit;
{
        struct xx *xxp = &xx[unit];
        struct ifnet *ifp = &xxp->xx_if;
        struct sockaddr_in *sin;
        int s;
        sin = (struct sockaddr_in *) &ifp->if_addr;
        if (sin->sin_addr.s_addr == 0)
                return;
        s = splimp();
        xxp->xx_oactive = 0;
         /* Initialize the hardware to receive packets */
        ifp->if_flags |= IFF_UP | IFF_RUNNING;
        if (ifp->if_snd.ifq_head)
                xx_start(unit);
        splx(s);
        if_rtinit(ifp, RTF_UP);
        arpwhohas(&xxp->xx_ac, &sin->sin_addr);
}
/*
 *      If the interface is not active, start output:
```

```
 *              dequeue a packet (a chain of mbufs) from the send queue
 *
 *              adjust the packet's length to ensure it is at least ETHERMIN
 *              bytes
 *
 *              if necessary, copy the data from the mbuf chain into the
 *              interface's private memory
 *
 *              free the mbuf chain
 *
 *              poke the device to start transmission
 *
 *              start a watchdog timer to make sure we notice if a
 *              transmission complete interrupt does not occur within a
 *              short time (in this case, two seconds)
 *
 *      Return value:  none.
 */
static
xx_start(unit)
 int    unit;
{
        int len;
        struct xx *xxp = &xx[unit];
        struct mbuf *m;
        if (xxp->xx_oactive == 0) {
                IF_DEQUEUE(&xxp->xx_if.if_snd, m);
                if (m == 0) {
                        xxp->xx_oactive = 0;
                        return;
```

```
                }
                len = /* packet length */
                if (len < ETHERMIN + sizeof(struct ether_header))
                        len = ETHERMIN + sizeof(struct ether_header);
                /* Copy from mbufs to interface memory (if necessary) */
                m_freem(m);
        }
        /* Do hardware-specific things to start packet transmission */
        xxp->xx_oactive = 1;
        xxp->xx_flags |= XX_TIMEOUTPENDING;
        timeout(xx_timeout, unit, v.v_hz << 2);
}
/*
 *      The transmit interrupt we expected has not occurred. Reset the
 *      device.
 *
 *      Return value: none.
 */
static
xx_timeout(unit)
 int unit;
{
        struct xx *xxp = &xx[unit];
        static int timeoutcount = 0;
        if (++timeoutcount > 100)
                printf("xx%d transmitter frozen -- resetting\n", unit);
        xxp->xx_flags &= ~XX_TIMEOUTPENDING;
        xx_init(unit);
}
/*
```

```
*      Transmit the packet in the mbuf chain m0 to dst using interface
*      ifp. Part of the handling of the packet is dictated by the
*      address family:
*      For IP packets:
*             compute the destination IP address
*
*             call arpresolve() to discern the destination's Ethernet address.
*             if arpresolve returns 0, the Ethernet address corresponding to
*             IP address idst is unknown, but arpresolve has taken charge of
*             the mbuf chain, so we indicate success.
*
*             set the Ethernet's packet type to ETHERPUP_IPTYPE
*
*             if this is a broadcast packet, and the interface is not capable
*             of receiving its own broadcasts, make a copy of the mbuf chain
*             so it can be passed to the loopback interface.
*
*      For raw Ethernet packets:
*             the destination address is expected to be an ether_addr
*             structure, containing the destination's Ethernet address
*             and packet type
*
*      Then:
*             set up the Ethernet header to be transmitted
*
*             enqueue the mbuf chain on the send queue; if the queue is full
*             drop the packet and free the mbuf chain, returning an error
*
*             if the transmitter is not currently active, start transmission
*
```

```
 *             if there is a packet to be fed back to the loop interface (if
 *             mcopy is not NULL), pass it to looutput
 *
 *      Return value:  if the packet was successfully enqueued on the
 *                     interface's output queue (and the loopback interface's
 *                     queue if this is a broadcast), 0 is returned.
 *                     Otherwise, the appropriate UNIX error number (see
 *                     <sys/errno.h>) is returned.
 *
 */
static
xx_output(ifp, m0, dst)
        struct ifnet *ifp;
        struct mbuf *m0;
        struct sockaddr *dst;
{
        int type, s, error;
        struct ether_addr edst;
        struct in_addr        idst;
        struct xx *xxp = &xx[ifp->if_unit];
        struct mbuf *m = m0;
        struct mbuf *mcopy = (struct mbuf *) 0;
        struct ether_header *e;
        switch (dst->sa_family) {
#ifdef INET
        case AF_INET:
                idst = ((struct sockaddr_in *) dst)->sin_addr;
                if (!arpresolve(&xxp->xx_ac, m, &idst, &edst))
                        return (0);
                type = ETHERPUP_IPTYPE;
```

```
                if (in_lnaof(idst) == INADDR_ANY)
                        mcopy = m_copy(m, 0, (int) M_COPYALL);
                goto gottype;
#endif
        case AF_UNSPEC:
                e = (struct ether_header *)dst->sa_data;
                edst = e->ether_dhost;
                type = e->ether_type;
                goto gottype;
        default:
                printf("xx%d: can't handle af%d\n", ifp->if_unit, dst->sa_family);
                error = EAFNOSUPPORT;
                goto bad;
        }
gottype:
        if (m->m_off > MMAXOFF || MMINOFF + sizeof(struct ether_header) > m->m_off) {
                m = m_get(M_DONTWAIT, MT_HEADER);
                if (m == 0) {
                        error = ENOBUFS;
                        goto bad;
                }
                m->m_next = m0;
                m->m_off = MMINOFF;
                m->m_len = sizeof(struct ether_header);
        } else {
                m->m_off -= sizeof(struct ether_header);
                m->m_len += sizeof(struct ether_header);
        }
        e = mtod(m, struct ether_header *);
        e->ether_type = htons((u_short) type);
```

```
        e->ether_dhost = edst;
        e->ether_shost = xxp->xx_enaddr;
        s = splimp();
        if (IF_QFULL(&ifp->if_snd)) {
                IF_DROP(&ifp->if_snd);
                splx(s);
                m_freem(m);
                return (ENOBUFS);
        }
        IF_ENQUEUE(&ifp->if_snd, m);
        if (xxp->xx_oactive == 0)
                xx_start(ifp->if_unit);
        splx(s);
        return (mcopy ? looutput(&loif, mcopy, dst) : 0);
bad:
        m_freem(m0);
        if (mcopy)
                m_freem(mcopy);
        return (error);
}
/*
 *      Interface interrupt routine. The argument is a structure one of
 *      whose members (a_dev) is the slot number of the interrupting
 *      interface. If this is a receive interrupt:
 *              if it was called by a receive error, increment the input
 *              error count
 *
 *              otherwise, call the receive interrupt routine
 *
 *      If this is a transmit interrupt:
```

```
 *             if it was called by a transmit error:
 *                    increment the output error count
 *                    mark the interface inactive
 *                    if there are packets on the send queue, restart output
 *
 *             otherwise, call the transmit interrupt routine
 *
 *      Return value:  none.
 */
xxint(args)
 struct args *args;
{
        int unit = xx_trans[args->a_dev - SLOT_LO];
        struct xx *xxp = &xx[unit];
        struct ifnet *ifp = &xxp->xx_if;
        int s;
        if (unit >= NXX) {
                printf("xxint: interrupt from slot %d\n", unit);
                panic("xxint");
                /*NOTREACHED*/
        }
        if (/* receive interrupt */)
                if (/* receive error */) {
                        /* reset hardware */
                        ifp->if_ierrors++;
                } else
                        xx_rintr(unit);
        if (/* transmit interrupt */)
                if (/* transmit error */) {
                        /* reset hardware */
```

```
                        ifp->if_oerrors++;
                        xxp->xx_oactive = 0;
                        s = splimp();
                        if (xxp->xx_if.if_snd.ifq_head)
                                xx_start(unit);
                        splx(s);
                } else
                        xx_xintr(unit);
}
/*
 *      Transmit interrupt routine:
 *              increment the count of packets transmitted
 *
 *              if there was a transmit timeout pending, cancel it
 *
 *              mark the interface inactive
 *
 *              if there are packets on the send queue, restart output
 *
 *      Return value: none.
 */
static
xx_xintr(unit)
 int    unit;
{
        struct xx *xxp = &xx[unit];
        int s;
        if (xxp->xx_oactive == 0)
                return;
        xxp->xx_if.if_opackets++;
```

```
        if(xxp->xx_flags & XX_TIMEOUTPENDING) {
                untimeout(xx_timeout, unit);
                xxp->xx_flags &= ~XX_TIMEOUTPENDING;
        }
        xxp->xx_oactive = 0;
        s = splimp();
        if (xxp->xx_if.if_snd.ifq_head)
                xx_start(unit);
        splx(s);
}
/*
 *      Receiver interrupt routine:
 *              increment the count of packets received
 *
 *              determine the Ethernet packet type and length, possibly
 *              dealing with ``trailer'' packets
 *
 *              call xx_get() to copy the packet from the interface's RAM
 *              into an mbuf chain and return a pointer to the first mbuf
 *
 *              pass the mbuf chain containing the packet to the appropriate
 *              input routine:
 *
 *                      for ARP packets, arpinput()
 *                      for reverse ARP packets, revarpinput()
 *                      for IP packets:
 *                              schedule a network software interrupt
 *                              if the input queue (ipintrq) is full, drop
 *                              the packet and free the mbuf chain; otherwise,
 *                              enqueue the packet on the IP input queue
```

```
 *
 *      Return value:  none.
 */
static
xx_rintr(unit)
 int    unit;
{
        short len;
        struct mbuf *m;
        struct ifqueue *inq;
        int s;
        u_short type;
        int off;
        int resid;
        caddr_t addr;
        struct xx *xxp = &xx[unit];
        struct mbuf *xx_get();
        /* Check for a received packet */
        xxp->xx_if.if_ipackets++;
        type = ntohs((u_short) /* packet type */);
        len = /* packet length */
        if (len < ETHERMIN || len > ETHERMTU) {
                xxp->xx_if.if_ierrors++;
                return;
        }
        addr = /* pointer to first byte in packet */
        /*
         *      Deal with trailer protocol: the ETHERTYPE_NTRAILER packet
         *      types starting at ETHERTYPE_TRAIL have (type -
         *      ETHERTYPE_TRAIL) * 512
```

```
         *      bytes of data followed by an Ethernet type and then the
         *      (variable-length) header
         */
#define         xx_dataaddr(addr, off, type)         ((type) (((caddr_t)((addr) + 1) +
(off))))
        if ((type >= ETHERPUP_TRAIL) &&
         (type < ETHERPUP_TRAIL + ETHERPUP_NTRAILER)) {
                off = (type - ETHERPUP_TRAIL) * 512;
                if (off >= ETHERMTU)
                        return;
                type = ntohs(*xx_dataaddr(addr, off, u_short *));
                resid = ntohs(*(xx_dataaddr(addr, off + 2, u_short *)));
                if (off + resid > len)
                        return;
                len = off + resid;
 } else
 off = 0;
#undef xx_dataaddr
        if (len == 0)
                return;
        /*
         *      Pull packet off interface. Off is nonzero if the packet has
         *      a trailing header. Xx_get() will then force the header
         *      information to be at the front, but we still have to drop
         *      the type and length which are at the front of any trailer
         *      data
         */
        m = xx_get(len, addr, off);
        if (m == 0)
                return;
        if (off) {
```

```
                m->m_off += 2 * sizeof (u_short);
                m->m_len -= 2 * sizeof (u_short);
        }
        switch (type) {
#ifdef INET
        case ETHERPUP_IPTYPE:
                schednetisr(NETISR_IP);
                inq = &ipintrq;
                break;
        case ETHERPUP_ARPTYPE:
                arpinput(&xxp->xx_ac, m);
                return;
        case ETHERPUP_REVARPTYPE:
                revarpinput(&xxp->xx_ac, m);
                return;
#endif
        default:
                m_freem(m);
                return;
        }
        s = splimp();
        if (IF_QFULL(inq)) {
                IF_DROP(inq);
                splx(s);
                m_freem(m);
                return;
        }
        IF_ENQUEUE(inq, m);
        splx(s);
        return;
```

```
}
/*
 *      Copy a packet of length totlen from the interface's RAM starting
 *      at buf. Off0 is nonzero if the packet is in ``trailer'' format.
 *
 *      Return value:  pointer to the first mbuf in the chain of mbufs
 *                     containing the packet.
 */
struct mbuf *
xx_get(totlen, buf, off0)
        u_char *buf;
        int totlen, off0;
{
        register struct mbuf *m;
        struct mbuf *top = 0, **mp = &top;
        register int off = off0, len;
        register u_char *cp;
        cp = buf;
        totlen -= sizeof(struct ether_header);
        while (totlen > 0) {
                MGET(m, M_DONTWAIT, MT_DATA);
                if (m == 0)
                        goto bad;
                if (off) {
                        len = totlen - off;
                        cp = buf + off;
                } else
                        len = totlen;
                if ((len < MCLBYTES) || (mclget(m) == 0)) {
                        m->m_len = MIN(MLEN, len);
```

```
                        m->m_off = MMINOFF;
                }
                bcopy(cp, mtod(m, caddr_t), m->m_len);
                cp += m->m_len;
                *mp = m;
                mp = &m->m_next;
                if (off0) {
                        off += m->m_len;
                        if (off == totlen) {
                                cp = buf;
                                off = 0;
                                totlen = off0;
                        }
                }
                else
                        totlen -= m->m_len;
        }
        return (top);
bad:
        if (top)
                m_freem(top);
        return (0);
}
/*
 *      Process an interface I/O control request. The only request the
 *      driver is currently expected to handle is SIOCSIFADDR (set
 *      interface address):
 *      We expect the pointer (data) passed to us to be a pointer to a
 *      sockaddr structure. We currently support two address families:
 *      AF_INET and AF_UNSPEC.
```

```
 *
 *      If we are passed an Internet address, we:
 *              call if_rtinit() to delete the previous routing table entry
 *              for this interface
 *
 *              call xx_setaddr() to set this interface's address
 *
 *              call xx_init() to reinitialize the software and hardware (it's
 *              possible this is the first call to xx_init after the interface's
 *              address has been set)
 *
 *      If we are passed a raw address (sa_family == AF_UNSPEC), we expect
 *      it to be an Ethernet address (an ether_addr structure) and set the
 *      device's hardware address, then call xx_init to reinitialize the
 *      software and hardware.
 *
 *      Return value:  if the I/O control is successfully completed, 0 is
 *                     returned. Otherwise, a UNIX error number (see
 *                     <sys/errno.h>) is returned.
 */
static
xx_ioctl(ifp, cmd, data)
        struct ifnet   *ifp;
        int            cmd;
        caddr_t                data;
{
        struct xx *xxp = &xx[ifp->if_unit];
        struct sockaddr *sa;
        struct sockaddr_in *sin;
        int s = splimp(), error = 0;
```

```
        switch (cmd) {
        case SIOCSIFADDR:
                sa = (struct sockaddr *) data;
                if (sa->sa_family == AF_UNSPEC) {
                        if (sa->sa_data[0] & 1) { /* broad or multi-cast */
                                error = EINVAL;
                                break;
                        }
                        xxp->xx_enaddr = *(struct ether_addr *)sa->sa_data;
                        xx_init(ifp->if_unit);
                        break;
                }
                sin = (struct sockaddr_in *)data;
                if (sin->sin_family != AF_INET) {
                        error = EINVAL;
                        break;
                }
                if (ifp->if_flags & IFF_RUNNING)
                        if_rtinit(ifp, -1);
                xx_setaddr(ifp, sin);
                xx_init(ifp->if_unit);
                break;
        default:
                error = EINVAL;
                break;
        }
        splx(s);
        return (error);
}
/*
```

```
 *      Record the interface's Internet addresses in the ifnet structure.
 */
static
xx_setaddr(ifp, sin)
        struct ifnet *ifp;
        struct sockaddr_in *sin;
{
        ifp->if_addr = *(struct sockaddr *) sin;
        ifp->if_net = in_netof(sin->sin_addr);
        ifp->if_host[0] = in_lnaof(sin->sin_addr);
        sin = (struct sockaddr_in *) &ifp->if_broadaddr;
        sin->sin_family = AF_INET;
        sin->sin_addr = if_makeaddr(ifp->if_net, INADDR_ANY);
        ifp->if_flags |= IFF_BROADCAST;
}
```

# Chapter 9

# Slot Device Drivers

A/UX was developed to make it easy to add slot devices, add-on cards that plug into the Macintosh II's six expansion slots. These cards use the Apple implementation of theNuBus protocol. A/UX requires a device driver for each card, regardless of the number of functions that the card supports. (This requirement may be different from cards developed for other operating systems.) Specific information about how slot ROMs are configured for the Macintosh II is found in *Developing Cards and Drivers for Macintosh II and Macintosh SE.*

## ROMs and autoconfiguration

Every slot device installed in the Macintosh II requires on-board ROM that provides module-specific system facilities to the A/UX system. The ROM supports module-specific resources residing in vendor-specified addressable NuBus memory, and presents a consistent interface to the running operating system or user programs.

When a system is booted, autoconfiguration searches the slots for devices and, if found, reads information contained in their slot ROMs. Before autoconfiguration can load a slot device driver, certain data structures found in the slot ROMs must be initialized with device information. For more information about these data structures and how to initialize them, see *Developing Cards and Drivers for Macintosh II and Macintosh SE.* Autoconfiguration is described in Chapter 12.

*Note:* During driver development, you may choose to install and test your driver without slot ROMs being present. Details about how to run autoconfiguration in this way are given in Chapter 13.

## The Slot Library

To make writing a slot device driver easier, A/UX supplies a set of routines called the Slot Library. The Slot Library provides a simple interface to the on-board ROM for each of the six Macintosh II slots. In Appendix C, you'll find descriptions of these routines, including the calling sequence, parameters, and return values.

There are three types of library routines: user routines, utility routines, and low-level routines. User routines can be called from user programs or kernel routines. Utility routines are used to gain access to slot ROM data structures, other resources, or other user programs. Low-level routines perform ROM access operations and operating system specific functions.

## Mapping to processes

There are two types of ROM base addresses: physical or virtual base. Slot ROM physical addresses are hexadecimal values having the following format:

```
0xFs0FF0000
```

where *s* is the NuBus slot number (9 to 14) of the board containing the ROM.

Slot ROM logical addresses are those that have been mapped from physical memory into user memory via the `phys`(2) system call. For user programs that use logical addresses, the slot number is a virtual address that corresponds directly to the physical address of the device.

## Interrupt service routines

Each slot controller card can generate one interrupt. The system then identifies the slot where the interrupt occurred and jumps to the appropriate driver code. There are no differences in the slots: you should be able to plug a card into any slot and have it work the same way.

Your device driver must supply an interrupt routine to service interrupts from your slot card. You specify that your device driver is a slot device driver and that your driver has an interrupt routine by including the flags `vs` in your master script file. (Chapter 12.describes the master script file.) These flags instruct `autoconfig` to add your driver interrupt routine to the slot interrupt vector table. For each slot card in the system, this table contains the address of the driver interrupt routine that services interrupts generated from that slot card.

When an interrupt occurs on your slot card, the kernel indexes the slot interrupt vector table and calls the routine stored at this address. The kernel passes a single parameter, called `args` (defined in `<sys/reg.h>`) to slot device driver routines. The kernel fills out various fields of this structure. In particular, the `a_dev` field of the `args` structure contains the slot number of the card that interrupted. This allows your driver to determine which of its slot cards interrupted. You can also use the slot number to determine the slot address space for the slot card.

The `autoconfig` utility creates an integer variable *prefix*`cnt`, which is an integer value containing the number of slot cards installed in the system that are controlled by your device driver. The `autoconfig` utility also creates a variable *prefix*`addr`, which is an array of integers containing *prefix*`cnt` elements. Each element of this array contains the slot number of a slot card installed in the system that is controlled by your driver .

See the chapter corresponding to your type of driver (Chapters 3 through 8) for details about how to write interrupt routines. The following page provides a quick reference description of the *driver*`int` routine of a slot device driver.

## Name

int—handle device interrupts from a slot device

## Synopsis

`void` ***driver*`int(`*args*`)`

`struct args *`*args*;

where

- *args* is a pointer to `a_dev` (the slot number). The `args` structure is defined in `<sys/reg.h>`.
- *driver* is the device prefix.

## Description

The interrupt routine of a slot device driver handles interrupts received from a slot device. The kernel passes a single parameter, the `args` parameter, to the interrupt routine of a slot device driver. You must give the kernel the address of your slot device driver interrupt routine during autoconfiguration. You do this by specifying the `vs` flags in your master script file.

*Note:* An interrupt routine should not change any variables in the u-dot or call `sleep()`.

## Return values

None.

## See also

For block drivers, see "The Block Device Interrupt Routine" in Chapter 2.

For character drivers, see "Handling Character Device Interrupts" in Chapter 4.

For information on the master script file,See "Using Device Information" in Chapter 12.

# Chapter 10

# SCSI Device Drivers

This chapter describes SCSI device drivers and how they gain access to a SCSI device. It assumes that you are familiar with the ANSI Small Computer System Interface (SCSI), the NCR 5380 SCSI chip,the Macintosh II architecture, and the A/UX device drivers described in this manual. For more information about the SCSI standard, see the Bibliography of this manual.

This chapter describes the SCSI manager in the A/UX Release 1.0. This chapter does not cover any changes or updates to the SCSI manager in later releases.

# An overview of the SCSI manager

The A/UX SCSI manager is a set of kernel software routines that device drivers use to gain access to the Macintosh II SCSI port. The main purpose of the SCSI manager is allow drivers to share the SCSI bus. In addition, the manager provides SCSI protocol handling and error notification. The SCSI manager simplifies programming of the chip and reduces the complexity of driver code.

Rather than having device drivers making single requests for low-level SCSI activities such as selecting the bus or requesting a status byte, the device driver creates a **request block data structure** that specifies the elements of a SCSI command, and passes this data structure to the SCSI manager.

The SCSI manager arbitrates for the SCSI bus and passes the request to the device via an NCR 5380 SCSI chip. The SCSI manager software performs I/O activity by reading and writing bytes during the various SCSI phases until the request has completed successfully or an error condition arises. The manager then notifies the device driver that the request is complete. Devices may disconnect from the bus during processing, then reconnect when processing has completed.

A SCSI device interaction is composed of three stages: a command, a read or write operation, and a completion sequence. The request block data structure contains pointers used during each of these operations.

Note that the SCSI manager obtains sense information from the device as part of the SCSI manager request and returns this sense information to the driver in the `sensebuf` field of the request block data structure. This ensures that the error information reflects the state of the SCSI bus when the failed transaction occurred.

## Assumptions and restrictions

The SCSI manager operates in a single-initiator environment: there can be only one initiator, the Macintosh II, on the SCSI bus. There can be up to seven other SCSI devices, numbered from 0 through 6, and each device can have up to eight logical units attached to it.

The A/UX operating system must enable interrupts before calling the SCSI manager. In particular, manager routines can't be called from driver routines that are invoked from a user-defined initialization routine.

The SCSI manager doesn't support devices that initiate messages when no request is outstanding. An unexpected message can't be passed to a device driver's attention routine.

The system may switch from an arbitrating system protocol (as described in the ANSI standard) to a single initiator system by waiting for arbitrated I/O transactions to complete. After arbitrated requests have stopped, single initiator requests can be issued. This is known as *exclusive use* in this document.

Remember that the SCSI bus is shared. Avoid increasing the performance of a single device while decreasing systemwide performance.

## Request block data structure

Drivers gain access to the SCSI manager by calling the routine `scsirequest()`, passing it two arguments: the SCSI ID of the device, and a pointer to a **request block data structure**. This data structure contains information about the request and allows the SCSI manager to process the request as a single action. The request block data structure is shown here:

```
struct scsireq {
        caddr_t         cmdbuf;         /* Buffer containing command block */
        caddr_t         databuf;        /* Buffer containing data to move */
        unsigned        datalen;        /* Length of the data buffer */
        unsigned datasent;              /* Length of data actually moved */
        caddr_t         sensebuf;       /* Result from sense cmd on error */
        int     (*faddr)();             /* Address of completion function */
        long    driver;                 /* Private storage for driver */
        struct scsireq link;            /* Link to next request */
        u_char cmdlen;                  /* Length of command buffer */
        u_char senselen;                /* Length of sense buffer */
        u_char sensesent;               /* Length of sense data received */
        u_short         flags;          /* Request flag bits */
        u_char msg;                     /* Message byte from completion */
        u_char stat;                    /* Completion status byte */
```

```
u_char ret;                    /* Return code from SCSI manager */
u_char timeout;                /* Maximum time for this request */
};
```

The fields of this structure are as follows:

- `cmdbuf` and `cmdlen` define the command to be sent to the device.
- `databuf` and `datalen` specifies the data area to be read or written from the device.
- `datasent` is where the SCSI manager returns the number of bytes actually transferred after the request has completed.
- `sensebuf` and `senslen` are set to receive the sense data before initiating a request.
- `senseset` is set to the actual number of bytes of sense data received, regardless of whether or not there is a buffer in the request to hold the data, if sense data is received. The driver should allocate memory space for all buffer areas and request data structures, since the SCSI manager does not perform memory allocation.
- `faddr` specifies the address of the completion service routine called when the SCSI request has completed. You must supply a completion service routine for your device. The routine is passed one argument, the request block pointer.
- `driver` is a 32-bit private storage area for the driver.
- `link` specifies execution of linked commands. The decision to link commands is left up to the driver. The driver should set the necessary bits in its command frame to tell the device that a sequence of linked commands is on the way. The SCSI manager continues sending commands until the chain has completed. If the device drops BSY, the SCSI manager repeatedly arbitrates for the bus and selects the device. A single interrupt is received either after an error occurs, or after all requests are processed. The request pointer passed back to the driver's interrupt handler reflects either the error, or the last request in the chain.
- `flags` controls aspects of the transfer. Bit values are OR'd together to fill this data field. Normally, it is data in or data out, but any 3-bit value can be specified. Possible uses for this field are discussed in "Special Processing," later in this chapter. Values for the `flags` field are as follows:

```
#define SRQ_READ 1                 /* This request read data */
#define SRQ_EXCL 2                 /* Exclusive SCSI use */
#define PRISHIFT 4                 /* Left shift for priority field */
#define SQR_PRIMASK (3<<4)/        * Four priority levels */
/* The remainder of request bits are not implemented */
#define SRQ_START 0x100            /* Call driver before request */
#define     SRQ_DATA 0x200         /* Call driver before data phase */
```

```
#define     SRQ_POST 0x400        /* Call driver after data phase */
#define     SRQ_NOSTAT 0x800      /* There will be no stat phase */
```

The SRQ_READ bit indicates that the expected data direction for this command is from the device to the computer. If there is no data phase expected for a command, the setting of this bit has no meaning. The SRQ_EXCL bit requests exclusive use of the SCSI bus for the next request. All outstanding disconnected I/O devices are allowed to reconnect and complete before the request is processed. There is a four-level priority scheme for requests, with jobs having numerically higher-priority levels being scheduled before lower priority levels. The SRQ_START, SRQ_DATA, and SRQ_POST bits request that the driver be notified at specific points during processing of the request (see "Special Processing" later in this chapter for more information).

- stat and msg return status and message bytes after normal completion of the request. SCSI devices indicate that more error sense information is available by turning on bit 1 in the status byte, which makes the SCSI manager execute a sense command.
- The ret field contains a return code from the SCSI manager, so it should be checked before the device status byte. The ret field reports request handling errors. Values for this field are as follows:

```
#define     SST_BSY 1       /* SCSI bus stayed busy */
#define     SST_CMD 2       /* Error during command */
#define     SST_COMP 3      /* Error during the status phase */
#define     SST_SENSE 4     /* Error obtaining sense data */
#define     SST_SEL 5       /* Nothing responded to ID */
#define     SST_TIMEOUT 6   /* Idle is longer than timeout value */
#define     SST_MULT 7      /* Multiple requests for this ID */
#define     SST_PROT 8      /* A problem in the SCSI protocol */
#define     SST_FATAL
#define     SST_MORE 9      /* More data than device expected */
#define     SST_LESS 10     /* Less data than device expected */
#define     SST_STAT 11     /* Error, sense command executed */
#define     SST_AGAIN 12    /* Place request again */
```

Error codes less than or equal to SST_FATAL are unusual. For these errors, multiple retries are not recommended.

Possible causes for error values of SST_TIMEOUT, SST_MULT, SST_MORE, SST_LESS, and SST_AGAIN include:

SST_TIMEOUT indicates that the driver disconnected, another device began a transaction and timed out, and the bus was reset to clear the other device. Thus, the device is left in an unknown state.

SST_MULT indicates that a second request was received for an ID that was currently processing a request. Remember that the device driver is responsible for coordinating multiple requests to a device.

SST_MORE indicates that the device changed phase before the buffer count reached zero.

SST_LESS indicates that the data count reached zero before the device changed phase.

SST_AGAIN indicates that another device has caused an error on the SCSI bus. Your driver's device has received a RST pulse, but was not the active device at the time of the error.

- timeout specifies the maximum number of seconds for the request. The SCSI manager rounds this value up to ensure that at least two watchdog timer intervals elapse. Currently a timer interval is 2 seconds, and it is rounded to 4 seconds. The maximum value for the timeout is 255 seconds, which is treated as infinite. Devices should modify the timeout field for long running operations, such as disk formatting.

## Other entry points and data structures

The following subsections describe the scsi_strings data structure and the scsig0cmd data structure. A following section describes the scsig0cmd routine.

### scsi_strings

The global array scsi_strings (defined in <sys/ncr5380.h>) contains error message strings indexed by manager return code. For example, the null-terminated string "scsi_timeout" is in position seven, indexed by SST_TIMEOUT. Use this array and the symbolic names for error codes to ensure that your driver can handle changes in error number assignments.

### scsig0cmd data structure

The data structure scsig0cmd (defined in <sys/ncr5380.h>) contains the command descriptor block sent to the controller. This structure is filled with values from the scsirequest structure and follows the ANSI format for SCSI commands. The scsig0cmd data structure is defined as follows:

```
struct scsig0cmd {

        u_char op;      /* 0: opcode */

        u_char addrH;  /* 1: logical address 2 and LUN */

        u_char addrM;  /* 2: logical address byte 1 */
```

```
        u_char addrL;  /* 3: logical address byte 0 */
        u_char len;    /* 4: number of blocks or bytes data */
        u_char ctl;    /* 5: control field */
};
```

where

- `op` is the operation code.
- `addrH` is the most significant byte of the logical block address(if required).
- `addrM` is the logical block address (if required).
- `addrL` is the least significant byte of the logical block address (if required).
- `len` is the transfer length (if required).
- `ctl` is the control byte.

### scsig0cmd routine

The `scsig0cmd` routine fills the `cmdbuf` array referenced by a `scsirequest` structure with a SCSI group zero command. The `scsig0cmd` routine is called as follows:

`int scsig0cmd(`*req, op, lun, addr, len, ctl*`)`

`struct scsireq *`*req*`;`

where

- *req* is the request parameter block. It must have a valid pointer to a `cmdbuf` data area that is at least six bytes long. The command is placed in this buffer and `scsig0cmd` sets the `cmdlen` data field to 6.
- *op* is the 8-bit opcode placed in byte 0 (see the ANSI standard for opcodes).
- *lun* is the logical unit number placed in the upper 3 bits of byte 1.
- *addr* is the 21-bit logical block address placed in bytes 1 through 3.
- *len* is the 8-bit transfer length placed in byte 4.
- *ctl* is the 8-bit control byte placed in byte 5.

### SCSI tasks

Each SCSI ID is a potential task. There is only one task outstanding per ID at any time, regardless of the number of logical units associated with an ID. The manager mantains a data structure for each ID indicating the task state, and a pointer to the current request for that ID. Each task is limited to having "legal" SCSI conversations (that is, those that follow the SCSI standard) with its device. These conversations have the following form:

*selection command data-in data-out status*

*selection* indicates a SCSI connection in which the computer tells the device that it can disconnect from the computer later on, as well as how to reestablish communication with the computer. *command*, *data-in*, *data-out*, and *status* correspond to SCSI COMMAND, DATA IN, DATA OUT, and STATUS phases, respectively. The device may signal a message phase at any time. Messages are not part of the semantics of legal conversations. Most commonly, a message indicates that the device is going to disconnect from the SCSI bus.

## Special processing

This section describes other entry points and methods of bypassing parts of the SCSI manager. Device-specific driver software can, under certain conditions, gain control prior to normal completion of processing. The request parameter block contains a pointer to a driver-specific interrupt function that will be called upon request completion. This function is not called from process context, so the contents of the kernel udot or upage data and stack area are undefined. Most importantly, a driver must not call `sleep()` during interrupt handling.

## Error handling

A watchdog timeout routine is scheduled continually at specified intervals. An 8-bit timeout field is found in each request block, and this field contains the maximum time that a device may remain inactive while processing a request. Drivers should be coded with increased timeout values if multi-block transfers are given as a single command, because any requested value less than 10 seconds is automatically increased to 10 seconds.

Error recovery on a timeout depends upon the state of the task. If the device is disconnected, the device must contend for the bus and an abort message sent. If the device is connected, the manager attempts to use the ATN line to send an abort message. If this doesn't work, match the device's phase and read junk values or write zeroes. ATN is kept high while waiting for the device to ask for a message. The device has 5 seconds to get off the bus after an abort begins. If a connected device doesn't drop BSY, or a disconnected device doesn't ask for a message, pulse the `SCSI RESET` line and notify the drivers. If in a data phase, the connected target receives an error saying that there was less data than the device expected, and all other targets also receive timeout errors.

If a disconnected device times out, other transactions are first allowed to complete, then the SCSI RESET line is asserted. If the active device times out, the manager asserts the ATN line. The device should ask for a message after which the manager will send an abort message. If the device ignores the ATN signal, the manager continues the transaction by reading ignored data or writing zeros.

## SCSI disk drivers

A typical SCSI disk driver can be divided into three layers. The top layer corresponds to routines called from the bdevsw table. The bdevsw routines typically check the minor number passed in the data structure describing the request with valid minor numbers for the device. Usually, the bdevsw routines take the parameters given to them, add a pointer to the device controller structure, and call generic routines, which comprise the middle layer. Generic disk software is driven from data structures that define disk access routines. These generic routines schedule I/O transactions and expand high-level requests, such as `ioctl`(2) calls, into the sequence of basic read and write requests needed. The generic routines often call device specific routines or low level routines to send the request to the actual hardware. The lowest layer implements simple, device-specific operations. They are sheltered from the details of processes, files, ioctls, and buffers.

The layers of a SCSI disk driver are illustrated in Figure 10-1.

**Figure 10-1**
SCSI disk driver

Each disk device has an associated data structure that describes its device and controller. This structure contains pointers to routines that perform simple functions such as reading, writing, or formatting.

To write a SCSI disk driver, you must do the following:

1. Determine how your disk hardware differs from SCSI command standards.
2. Replace those routines in the generic library with device-specific ones for your driver.
3. Arrange for your driver to be autoconfigured into the A/UX kernel.

Appendix G contains a listing of the source to an A/UX SCSI device driver.

## Device naming conventions

Named file entries in the `/dev/dsk` directory contain SCSI entries labeled c*n*d*n*s*n* where *n* is a decimal number assigning the controller, device, and slice. The controller number is the logical bus ID of the SCSI device (0-7), the drive number is a logical unit number (0-7), and the slice is the logical partition number (0 to 31).

The device is named according to the convention used in the stand-alone code and the kernel, although device drivers actually recognize devices by their major and minor numbers. For SCSI disk devices, the major number determines the SCSI controller bus ID (eight consecutive major numbers correspond to SCSI IDs 0 to 7), and the minor number determines the logical unit number and partition, as shown in Figure 10-2.

**Figure 10-2**
Minor number assignment

This scheme allows 32 partition numbers. Each partition is dynamically assigned by a partition map and hundreds of named partitions can be on a single disk.

## Disk partitioning

The disk partition map data structure provides a 32-character name for each partition. Named partitions are associated dynamically with numbered devices through `ioctl`(2) calls. Default values are assigned to the first 16 partitions at boot time. The first three partitions (0–2) are assigned to the default root, swap, and usr file system. If any of these three file systems are missing on the current disk, then the partition number is unassigned. The next 13 file systems (3–15) are assigned in order of the file systems on the disk. If the active root, swap, or usr file systems are among the first 13 partitions on the disk, the second occurrence of the file system is left as an unassigned partition.

In addition to these assignments, the final partition, 31, always maps to the entire physical disk. Any user program may read from this partition (assuming an inode with the appropriate permissions is available), although the device driver only grants write access to this partition to programs running with superuser privileges.

For details about the Macintosh II disk partitioning scheme, see *Inside Macintosh, Volume 5.*

## Typical I/O operation

A typical I/O operation begins when a read or write call occurs in the context of a requesting process. Typically, the driver will be active and will schedule the request for a later time. When the request actually runs, the operating system might be processing an interrupt outside the context of the requesting process. The sequence of device driver calls to place a request is shown in Figure 10-3.

**Figure 10-3**
Initiation of typical I/O request

After the strategy routine places the I/O request into the device's queue, the generic start routine schedules the request, as shown in Figure 10-4.

**Figure 10-4**
I/O request processing outside process context

Only one outstanding request per controller is allowed. The external device interrupts the CPU to signal that the request has finished. The interrupt can call the device handler directly for a slot-based device, or the device handler may be called after the SCSI manager is initially called. Device-specific code responds to the interrupt, determines if the transfer completed without error, and calls a request completion routine in the generic code. The generic request completion routine informs the rest of the operating system that the request has completed. In standard drivers, no driver-specific code is executed in process context while servicing an interrupt. Generic code may, however, arrange for a sequence of sleeps and wakeups to read a partition map for an ioctl call, for example. Handling of specific calls is discussed in more detail in the section "Generic Routines".

## Data structures on disk

A portion of the disk reserved for A/UX is defined by the **disk partition map entry** (dpme) and block zero block data structures.All operating systems using the disk share Apple's disk partition map entry format, but a driver can gain access to partitions belonging to A/UX only.

*Note:* The A/UX utility dp(1e), which performs disk partitioning, can be used to create and change disk partition information.You can also use the utility Apple HD SC Setup 2.0, which is documented in a preliminary note available through APDA.

The dpme data structure contains fields defining the logical start address and the number of blocks, which define the area of a partition that contains a partition. Normally, the end of the partition (that is, *past start address + number of blocks*) contains an optional spare block area used for bad block handling. The fields of the dpme data structure are listed in dpme(4).

The dpme data structure reserves space for operating system specific information. For A/UX, this space is called the **block zero block** (bzb) data structure. The driver modifies and updates several of these fields. The driver uses the bzb data structure to assign file partitions to eschatology clusters and to determine the position of the alternate block map.

A single disk might have several root file systems. Each may be a cluster of file systems that contains its own root, usr, swap, and eschatology backup file systems. The generic driver code obtains the number of the default eschatology cluster from the Apple boot-up firmware, which makes sure that the default root, swap, and usr file systems are mapped as minor devices zero, one, and two.

The alternate block map (abmh) data structure consists of a header, followed by a variable length list of block numbers, as shown here:

```
struct abmh {

        u32     abmh_magic
```

```
        u32     abmh_len;
};
#define                 ABMH_MAGIC      0xBABEEEE
```

where

- `abmh_magic` is a magic number
- `abmh_len` is the length of the block number list.

The block number list is an array of long integers. Each indexed location in the array corresponds to a potential alternate block in the spare block area. The location in the alternate block array can contain either the number of a block in the data portion of the disk partition that will be remapped, or a flag value. Possible flag values are as follows:

–1 Blocks available

–2 Bad free block—do not use

–3 Block allocated to alternate block map

You cannot make any assumptions about the ordering of bad block information on the disk.

*Warning*: Never offset the logical data area of the partition from the start of the physical partition. Although driver code allows this, block numbers of blocks cached in core would then be incorrect.

## Kernel data structures

Three levels of data structures describe disks:

- controller
- drive
- partition

Controller-level data structures define methods of accessing the disk and define the software that is called for each access. A controller corresponds to a SCSI ID and to a major device number and any given controller may be present on several IDs.

Drive-level data structures describe drives connected to a controller. Several drives may be attached to a given SCSI controller.

Partition-level data structures describe partitions on a drive. A drive is divided into a number of minor devices. At any moment, a minor device may or may not be assigned to a partition on the disk. When a minor device is associated with a partition, there is a device partition map entry, bad block information, and user data available. When a minor device is not associated with a partition,you can perform open(2), `close`(2), and some `ioctl`(2) calls, but performing `read`(2) and `write`(2) calls return errors.

The generic driver open routine allocates these disk data structures as needed using the `kmem_alloc` memory manager routine. Controller data structures are never freed. The pointer to the controller data structure is not associated with the device switch information for the device; the high-level device-specific code must keep track of the pointer. Dynamic allocation ensures that unused data structures do not consume space.

### Controller data structures

There is one controller data structure for each controller. All drives having the same major number use the same data structure. The controller data structure is shown next.

```
struct gdctl {                          /*     Controller data structure */
        struct genprocs *ctprocs;          /* generic low level procs */
        struct gentask  *cttaskp;          /* pointer to current task list */
        int     ctflags;                   /* flags for handling controller */
        struct gddrive *ctdrive;           /* drive list */
        struct gddrive *ctactive;          /* currently active drive */
        struct gdctl *ctnextct;            /* list of ctl structures */
        int     ctcmd;                     /* Command associated with ctbp */
        long    ctarg;                     /* argument for current command */
        int     (*ctdevctl)();           /* function to be called for devctl */
        int     ctretval;                  /* return value of command */
        daddr_t ctsector;                 /* private for generic code */
        daddr_t ctlbn;                     /* logical block for error msgs */
        struct  buf *ctbp;                 /* allocated scratch buffer */
        struct  deverreg cterr[4];         /* Scratch for error messages */
        short   cterrind;                /* index into error message storage */
        char    ctrunning;                 /* True if start routine active */
        char    ctpending;               /* True if any device has a request */
        short   ctstate;                   /* generic code private data */
        short   ctretry;                   /* retry counter for soft errors */
        short   ctmajor;                   /* major device number from devsw */
        };
```

where

- `ctprocs` points to an array of entry point addresses for device-specific routines that perform specific tasks (see the `genprocs` data structure described later).
- `cttaskp` points to the current task data structure (see the `gentask` data structure described later).
- `ctflags` holds various controller state flags. Possible states are

  | | |
  |---|---|
  | `NOPRINT` | If set, console error printingis supressed. |
  | `CLOSING` | Set by generic code while device is closing. |

- `ctdrive` is a list of drives associated with this controller (see the `drive` data structure described later).
- `ctactive` points to the currently active driver from the `ctdrive` list.
- `ctnextctis` a linked list of all controllers in the system. The generic code uses this list to locate a controller associated with a given major device.
- `ctcmd` is the command associated with the controller buffer. It contains a code for a currently queued ioctl (see "Controller Data Structures" later in this chapter).
- `ctarg` is an optional argument associated with the device-specific ioctls being passed from the high-level to the low-level device-specific code.
- `ctdevctl` is a device specific function that the generic routines call to initiate exclusive control functions. `ctdevctl` is responsible for calling `gdrestart`.
- `ctretval` is used to return the completion status from device-specific routines to generic routines.
- `ctsector` is a private location for generic routines.
- `ctlbn` is used by generic routines which place the block number in this in anticipation of diagnostics.
- `ctbp` is a buffer from the buffer pool assigned to the controller when the drive is first opened. It remains assigned for the duration of A/UX execution. The buffer space is used to read partition information for device initialization and ioctl processing. The buffer header provides concurrency control for device ioctl processing.
- `cterr` is a scratch location for `gderr` logging of error messages.
- `cterrind` is a scratch location for `gderr` logging of error messages.
- `ctrunning` is a flag which, if `TRUE`, indicates that there is an outstanding request for any drive on the controller.
- `ctpending` is a flag which, if `TRUE`, indicates that a drive for this controller has a queued request. This flag is cleared when the interrupt handler has finished handling requests.
- `ctstate` is a state variable for the controller that organizes activities across interrupts. For example, it could increment this field from one to four for a four-step initialization sequence.
- `ctretry` is a private counter for generic retrying of requests.

- `ctmajor` is the major number of the device.

The `genprocs` data structure defines low-level device-specific procedures that are called to process specific requests. Each entry point is a pointer to a function. The specific entry points are described in detail in "Low-Level Device Routines", given later in this chapter. The data structure is created when the device is first opened, then is initialized by the generic code to point to SCSI routines. Device-specific code can then modify the entries.The `genprocs` data structure is shown here:

```
struct genprocs {
        int     (*gpread)();            /* Read into buffer */
        int     (*gpwrite)();           /* Output buffer to device */
        int     (*gpdriveinit)();       /* Initialize data structures */
        int     (*gpbadblock)();        /* Map bad block */
        int     (*gpformat)();          /* Format drive */
        int     (*gprecover)();         /* Recover following an error */
        int     (*gpshutdown)();        /* Stop processing or eject */
        };
```

The task data structure, `gentask`, describes one I/O operation. One data structure exists per controller. For more about this structure, see "Low-Level Device Routines", given later in this chapter. The `gentask` data structure is shown here:

```
struct gentask {
        int (*gtretproc)();             /* Address of completion function */
        struct gdctl *gtctp;  /         * Pointer back to controller */
        struct drqual *gtqual;          /* Device qualities pointer */
        caddr_t gtaddr;                 /* Address of buffer to fill */
        int gtnreq;                     /* Number of bytes requested */
        int gtndone;                    /* Number of bytes read/written */
        daddr_t gtblock;                /* Block number to read or write */
        short gtmaj;                    /* Device major number */
        short gtdnum;                   /* Disk number */
        };
```

### Drive data structures

Drive data structures are created as needed when a drive is opened. There is one or more drives associated with a controller and each drive is normally a single spindle. The space allocated is never released. The `gddrive` data structure is shown here:

```
struct gddrive {                    /* description of a single spindle */
        struct gddrive *drnxt;          /* next drive on controller */
        struct drqual *drqual;          /* qualities of device */
        short   drstate;                /* drive state from generic code */
        short   drpartnum;          /* which partition are we working on */
        int     drcount;                /* count for EOF calculation */
        u_char  drnum;                  /* device unit number */
        struct  iobuf drtab;            /* I/O queue header */
        struct  iostat driostat;        /* I/O error handling */
        struct gdpart *drpart[GD_MAXPART];  /* pointers to partition info */
        };
```

where

- `drnxt` is the next drive in the linked list of drives associated with this controller.
- `drqual` is the device-qualities data structure describing this drive. The `drqual` data structureis described later in this section.
- `drstate` is the drive state. Possible states are

  | | |
  |---|---|
  | `NOTINIT` | The drive has never been accessed. |
  | `REINIT` | Must be initialized again on next access. |
  | `STARTING` | The drive is in the process of being initialized. |
  | `NORMAL` | The drive is ready. |

- `drpartnum` is a partition number in the range 0 to 31.
- `drcount` is the count used during end-of-file calculations.
- `drnum` is the device number or SCSI logical unit number of the drive. This number is always the upper 3 bits of the 8-bit minor device number.
- `drtab` is the A/UX I/O queue header.
- `driostat` is the I/O statistics and error handling data.
- `drpart` is an array of 32 pointers to partition structures.

Each drive has a group of qualities that define the drive. The data structure that defines these qualities is shared between the generic code and the device-specific code. The qualities data structure, `drqual`, is made available to the device-specific code on each I/O operation. The `drqual` data structue is shown here:

```
struct drqual {

        long dqdevp;            /* device-specific storage */

        int dqflags;            /* Flag values */
```

```
        u_long dqxfermax;       /* Maximum number of blocks per transfer */
        short dqcyl;            /* Number of sectors per cylinder */
                                /* (negative for no cylinder boundaries) */
        u_short dqblksize;      /* Currently unused */
        daddr_t dqmaxbn;        /* Maximum block number for device */
};
```

where

- dqdevp is a location used for device-specific storage. Device driver routines may use it for any purpose, becausethe generic code will never modify this location.
- dqflags is a bit array of flags.
- dqxfermax is the maximum transfer size, which is the largest number of bytes that should be sent to the device in a single request. The generic code breaks large read requests into "chunks" no larger than this size.
- dqcyl is the number of sectors per cylinder. This field is used for error messages. It is available to applications such as mkfs via an ioctl. If this number is positive, then long requests are broken on cylinder boundaries; if negative, dqcyl is the cylinder size, but requests are not broken on cylinder boundaries.
- dqblksize is the physical block size of the device.
- dqmaxbn is the maximum block number of the device. This number reflects the size of the device, ignoring disk partitioning or reserved areas.

### Partition data structures

Each drive can support 32 active partitions at a time. Partition data structures are created as needed, and the space they occupy is never released. The partition data structure is shown here:

```
struct gdpart {                 /* Description of a mounted partition */
        long ptflags;           /* Various flags */
        daddr_t ptdpme;         /* Disk address of dpme entry */
        daddr_t ptoffset;       /* Physical address of first block */
        daadr_t ptlsize;        /* Logical size of data partition */
        daddr_t ptpsize;        /* Total size of partition */
        daddr_t ptastart;       /* Location of alt block map */
        int ptasize;            /* Size of alt block map (in bytes) */
        int ptaents;            /* Number of entries in alt block (in bytes) */
```

```
short ptstate;        /* State information */
struct bbhdr *ptbm;   /* Bad block bucket list for partition */
short ptbmask;        /* Mask for bucket hashing */
char ptname [32];     /* Name of partition */
char pttype (32];     /* Name of partition */
char ptcluster;       /* Eschatology cluster of partition */
};
```

where

- `ptflags` is the flags from the set. Possible values are

  Partition flag values:

  | | |
  |---|---|
  | `NOALT` | Alt block mapping is disabled for the partition. |
  | `USERALT` | The user has explicitly turned off alt block mapping. |
  | `ESCH0` | Partition is default autorecovery cluster. |
  | `RONLY` | Read-only file system (for example, CD-ROM). |

  Partition type values:

  | | |
  |---|---|
  | `TYPHYS` | Partition is whole device, not partition. |
  | `TYDEF` | Default partitioning was supplied. |
  | `TYDPME` | Partition assigned from DPME. |

  Partition name values:

  | | |
  |---|---|
  | `NMNONE` | This number has no partition assigned. |
  | `NMUSER` | The name/number assigned by user ioctl. |
  | `NMDEF` | The name/number assigned by default. |

- `ptdpme` is the disk address of the disk partition map entry for this partition.
- `ptoffset` is the physical address of the first block of data.
- `ptsize` is the size of the data area of the partition.
- `ptstate` is the state information for the partition. Possible values are

  | | |
  |---|---|
  | `REINIT` | The partition information must be reinitialized on the next read or write access. The read or write will fail unless the partition is assigned a name by default, or by an application's ioctl routine |
  | `STARTING` | The driver is in the process of initializing the partition. |
  | `NORMAL` | The partition is initialized and ready. |
  | `NEEDALT` | Alt block processing is required. |
  | `ALTING` | Alt block processing is in progress. |

- `ptbm` points to the beginning of the bad block hash list.
- `ptbmask` is the mask for locating bad blocks in the bad block hash list.
- `ptname` is the name of the partition set by Apple's administrative software. The name is a null-terminated string.

Bad block information is associated with each partition. See "Bad Block Handling" in this chapter for more information.

## Generic routines

The generic device driver routines provide a layer of subroutines between the code called from entries in the bdevsw structureand the device-specific code that gains access to the hardware. The high-level device-specific code is called directly from the bdevsw table and passes a request on to the generic routines. The generic routines, in turn, enqueue requests and pass them to the low-level device-specific routines. The generic disk driver implements open, close, strategy and ioctl services. In addition, the software maintains a disk partition map and alternate block mapping for a device. The generic driver is closely attuned to the requirements of SCSI disks, but it can also be used with other controllers as well.

The routines described next provide the interface to the generic driver. The generic open, close, strategy, and ioctl routines are described in this section. Note that there are no generic entry points for unbuffered reads or writes from character special drives. Normally, high-level disk read and write routines call `physio` with the address of the strategy routine. When using the generic disk driver, the strategy routine passed to `physio` is device-specific code, which in turn calls `gdstrategy`.

The gdopen routine is called as follows:

`gdopen` (*ctp, dev, flag*)

`struct ctl` **ctp*;

`dev_t` *dev*;

`int` *flag*;

where

- *ctp* is the controller data structure for this major number.
- *dev* is the device number
- *flag* is a read/write flag.

The gdopen routine opens a drive. If the drive has never been opened, this routine creates the appropriate drive and partition structures. The device-specific code must have previously called `gdinit` on the first open of this major number. Opening a device doesn't check that the device is ready for access. In particular, you can open an improperly formatted drive and format it, which can result in delayed notification of common errors until the first read or write.

The `gdclose` routine is called as follows:

```
gdclose (ctp, dev)

struct ctl *ctp;

dev_t dev;
```

where

- *ctp* is the controller data structure for this major number.
- *dev* is the device number.

The `gdclose` routine closes a drive. Closing a device has very little effect. The low-level shutdown routine is called with an argument indicating that the device was closed. Any partitions associated with the device remains associated with it on the next open.

The `gdstrategy` routine is called as follows:

```
gdstrategy (ctp, bp)

struct ctl *ctp;

struct buf *bp;
```

where

- *ctp* is the controller data structure for this major number.
- *bp* is the `buf` structure that describes the I/O request. The `buf` structure contains the address of the buffer associated with the I/O request.

The strategy routine places a buffer in the drive's queue for a later I/O operation. Each controller has a scratch buffer header, pointed to by `ctbp`, which is assigned on the first open. The controller buffer is used to schedule ioctl and other control operations. The ioctl routine waits for the controller buffer using the normal sleep and wakeup mechanisms. When this buffer becomes available, the controller data locations `ctcmd` and `ctcmdarg` also become available; only the process that has exclusive use of the controller buffer may write to these locations. The ioctl routine then passes the address of the controller buffer to `gdstrategy` which treats this buffer as a special case, and is placed at the end of the queue. When the start routine finds the command buffer at the head of the queue, it takes the necessary steps to process the ioctl.

The `gdioctl` routine is called as follows:

`gdioctl (`*ctp, dev, cmd, addr, flag*`)`

`struct ctl *`*ctp*`;`

`dev_t` *dev*`;`

`int` *cmd*`;`

`caddr_t` *addr*`;`

`int` *flag*`;`

where

- *ctp* is the controller data structure.
- *dev* is the device number.
- *cmd* is the ioctl command code (The command codes are documented in gd(7)).
- *addr* is the address of the ioctl call arguments.
- *flag* contains the flags associated with the file.

This routine implements the generic set of SCSI ioctl calls. It recognizes the ioctl number and performs the necessary actions. If it doesn't recognize the ioctl type, it returns an error to the user. The list of ioctls is found in gd(7). Also, you can use the `gddevctl` routine for extendable device-specific ioctls, as described next.

The `gddevctl` routine is called as follows:

`gddevctl (`*ctp, dev, dproc, arg*`)`

`struct ctl *`*ctp*`;`

`dev_t` *dev*`;`

`int (*`*dproc*`)();`

`int` *arg*`;`

where

- *ctp* is the controller data structure for this major number
- *dev* is the device number.
- *dproc* is the ioctl process to schedule.
- *arg* is the argument passed to the process.

This routine provides a method for your driver to perform hardware-specific ioctls. The high-level device-specific ioctl routine is called from the `bdevsw` table, where it is examined and either passed to the generic code or acted on. If the driver-specific code must take action, the device code must wait until the hardware is not busy.

When the device is ready to process the ioctl, the device-specific routine given as *dproc* is called with the arguments (`*dproc`) (*major, unit, arg*). dproc is not called from process context. It is able to gain access to use data space. The high-level device-specific ioctl routine is expected to sleep and arrange to be awakened so that it can carry data or status back to the user. The high-level routine, in process context, and routines called from *dproc* can arrange via the buffer pool, to move data. *dproc* can call low-level device-specific routines, or arrange for hardware interrupts as required. Until it relinquishes control, it has exclusive use of the hardware. When all processing for the ioctl has completed, the completion routine `gdrestart` should be called.

## Service routines for device-specific code

The routines described in this section provide services for device-specific code. The device-specific code is responsible for keeping track of a pointer to the controller data structure. The `gdctlinit` routine creates the controller data structure when the first access to it occurs. The `gdctlinit` routine is called with the following parameters:

`struct ctl *  gdctlinit(`*major, minor*`)`

`int` *major, minor*;

where *major* and *minor* are the A/UX major and minor numbers.

The device-specific code must keep track of a pointer to the controller data structure. The `gderr` routine, shown here, creates the controller data structure on the first access.

`gderr(`*taskp, str, num*`)`

`struct gentask *`*taskp*;

`char *`*str*;

`int` *num*;

where

- *taskp* is the request being serviced.
- *str* is the error message string.
- *num* is the number associated with the error.

The gderr routine hides the data structures that interface to the error message handler. Low-level interrupt service routines can make repeated called to this routine. When the generic code's interrupt completion routine is called, a System V-style error message is formatted and passed to logberr.

The gdrestart routine is called as follows:

```
gdrestart(major, reinit)
        int major;
        int reinit;
```

where

- *major* is the device to restart processing.
- *reinit* is an integer flag that, if TRUE, tells the driver to set everything to reinitialization state.

gdrestart is called after the processing of a device-specific I/O call initiated from gddevctl. It ends a period of exclusive use for the device. If the *reinit* flag is TRUE, all drives and partitions associated with the controller will be set to REINIT. This will not disturb any recently queued I/O.

## Low-level device routines

The low-level routines (also called *procs*) perform simple hardware-dependent operations. Most SCSI disk devices use the same read and write routines, but vary in the way they handle options such as bad block handling. The low-level routines are sheltered from the exact details of controller, driver, and bad block handling.

The low-level routines are specified in the genprocs data structure (see "Kernel Data Structures"), which the generic code initializes to values appropriate for a generic SCSI disk. The high-level device-specific code can reset any of the values as needed.

Upon completion of I/O, the low-level routine calls the return address given in the gentask data structure. The return address varies depending upon the state of the generic device model. The callback function expects two arguments: the task pointer and the return status. The return status values and what they indicate are as follows:

| | |
|---|---|
| GDR_OK | The operation was a success. |
| GDR_AGAIN | A correctable error occurred, so you should perform the request again. Partial data in the buffer might be intact. There may be an error message data to log. |
| GDR_CORR | A correctable error occurred and has been corrected. There may be an error message to log. |

`GDR_FAILED` The operation failed. Partial data in the buffer might be intact. Your driver should have supplied an error message to log.

The device qualities (`drqual`) data structure includes space where the device code can keep a pointer to private data structures that further describe the device. The same `drqual` data structure is presented each time a driver gains access to a given controller and drive.

The entry points to the low-level device routines are given next. For each low-level routine, *taskp* is a pointer to a task data structure.The entry point for the low-level read routine is shown here:

`d_read(`*taskp*`)`

This routine moves *n* bytes of data from the disk location indicated by `gtblock` to the buffer pointer to by the task pointer. The device routine updates the `gtndone` field of the task structure.

The entry point for the low-level write routine is shown here:

`d_write(`*taskp*`)`

This routine moves *n* bytes of data from the buffer pointer to by `gtaddr` to the block pointed to by gtblock. The routine updates the `gtndone` field.

The entry point for the low-level initialization routine is shown here:

`d_driveinit(`*taskp*`)`

This routine initializes the device qualities data structure. The device-specific field of this data structure is NULL if this is the first time `d_qualinit` has been called for this drive. A drive might be initialized repeatedly as part of error recovery, formatting, or other ioctls. The address pointed to by the gtaddr field of the task pointer is the controller scratch buffer.

The entry point for the low-level bad block handling routine is shown here:

`d_badblock(`*taskp*`)`

This routine asks the device to mark the block `taskp-> gtblock` as bad. The address pointed to by the gtaddr field of the task pointer is the controller scratch buffer.

The entry point for the low-level formatting routine is shown here:

`d_format(`*taskp*`)`

This routine formats the drive. The previous contents of the disk will be lost. The address pointed to be the `gtaddr` field of the task pointer is the controller scratch buffer.

The entry point for the low-level reset routine is shown here:

`d_reset(`*taskp*`)`

The generic code calls this routine following any uncorrected error. If this routine returns an uncorrected error code, the drive is marked as DOWN. The address pointed to by the `gtaddr` field of the task pointer is the controller scratch buffer.

The entry point for the low-level shutdown routine is shown here:

`d_shutdown(`*taskp*`)`

`taskp->gtnreq` indicates that a shutdown value was passed from the user's ioctl. Only two values are currently defined. 0 means that the device should retract its heads to prepare for shipping, and 1 means that a partition on the unit has just closed. The address pointed to by the gtaddr field of the task pointer is the controller scratch buffer.

# Chapter 11

# Apple Desktop Bus Drivers

If you're adding a device that uses the Apple Desktop Bus (ADB), you should read this chapter before you begin. While ADB device drivers use the same interfaces to A/UX processes as other drivers do, special support is required to share the ADB with other drivers. In this chapter, the Apple Desktop Bus (formerly Front Desk Bus) routines, files, and commands use the prefix `fdb`.

The Apple Desktop Bus (ADB) is a simple serial bus used to access peripheral devices such as keyboards and mouse devices that are usually located on your desktop. The ADB takes multiple ADB requests from system software, sends them to their appropriate devices, and returns the results to the same software that requested them. It allows the system to poll individual devices for state changes and notify the system of such state changes. Because of the simple hardware, a single kernel interface is required that must do the following:

- Serialize ADB transactions, because only one transaction can be run at a time. Pending transactions are stored and run in a round-robin manner.
- Support interrupts. Each ADB transaction encounters several hardware interrupts before the transaction is complete. The device driver only needs to make a request to receive a reply later—interrupts are handled transparently for the driver.
- Support hardware polling. The ADB controller chip periodically repeats the last read transaction executed on the ADB bus. If such a hardware poll is successful, then the appropriate driver is notified of the successful poll and the data is returned.
- Support software polling. When a device on the ADB requests service, the system is interrupted. Because the hardware provides no mechanism to determine which device is making the request ,the ADB support software asks each known device driver to poll its corresponding device to see if service is requested. If the service request is removed by one device driver's polling, any other outstanding software polls are canceled and their drivers are notified.

## Transactions

A **transaction** is the basic function requested of the ADB. A transaction consists of a request for the ADB software from a driver, an action, and a reply from the ADB software after the action has completed. A transaction is always specific to a particular device (with a particular address on the ADB). A particular device may have only one transaction outstanding at a time.

The ADB supports three basic types of transaction requests:

`flush(`*device*`)` This is used to instruct the device to flush itself (for example, to empty its internal buffers of stored keystrokes for a keyboard).

`talk(`*device, register*`)` This is used to read from a register on the device. This transaction is an instruction to the device to "talk" to the system. A device contains four registers numbered 0 to 3. If a timeout does not occur, a talk request returns data read from the contents of the register.

`listen(`*device,register, data*`)` This is used to write to a register on a device. This transaction is an instruction to the device to "listen" to the system. A listen command must include the data to be written.

## Driver service routines

When the ADB first grants access to each driver, it must provide the ADB software with the address of an interrupt service routine (see "Initiate Transaction" in the next section). The ADB software calls this service routine at the end of each ADB transaction to pass back data and to notify the driver that the transaction is completed. This routine is also called when certain exception device polling conditions exist.

A driver service routine is always called with three integer parameters. The first parameter is the ID number specified when the transaction was started. The second parameter is a value that specifies what type of transaction has completed (called the *command*), and the third parameter (called the *arg*) is command specific. Symbols for the command values are found in the file `<sys/fdb.h>`.

## High-level driver routines

High-level drivers can call the routines provided in the ADB kernel code to perform ADB transactions. This section describes these routines.

### Initiate transaction

The `fdb_open` routine makes the first transaction to a device on the ADB. It is usually called once from a device driver's init routine (when the system is initialized).You can call the `fdb_open` routine as follows:

`fdb_open(`*addr, id, intr*`)`

where

- *addr* is the address of the device that is being accessed on the ADB (a number in the range 0–15).
- *id* is a number (usually the device's minor number) that is returned with the transaction's completion indicator.

- *intr* is the address of the device's interrupt service routine. The bus software calls this routine at the end of each ADB transaction to pass back data and to tell the driver that the transaction is completed. The device's interrupt service routine is also called when certain device polling exceptions occur.

In addition, `fdb_open()` initiates an ADB transaction (actually a talk to register 3 of the device) that determines if the device really exists. When this transaction completes, it always calls the service routine with the *command* `FDB_EXISTS`. In this case, the *arg* parameter can have two possible values: 0 if the device really does exist on the ADB, or nonzero if a timeout occurred while trying to talk to the device, and the device is not present on the bus.

## Flushing a device

The `fdb_flush` routine flushes data from a device. You can call the `fdb_flush` routine as follows:

`fdb_flush`(*addr, id*)

where

- *addr* is the bus address of the device being flushed.
- *id* is a number passed back to the driver with the service routine.

When the flush transaction completes, the device's service routine is called with the *command* `FDB_FLUSH`. If the *arg* is nonzero, then a timeout occurred and the device is not present on the bus.

## Talking to the system

The `fdb_talk` routine instructs a device to "talk" to the system. You can call the `fdb_talk` routine as follows:

`fdb_talk`(*addr, id, register, datap*)

where:

- *addr* is the address of the device where the talk is initiated from.
- *id* is a number to be passed back to the driver with the service routine.
- *register* is the register being read (talking), and is a value between 0 and 3.
- *datap* is the address of the buffer to contain the data being read.

When the talk transaction is completed, the device's service routine is called with the *command* FDB_TALK. If nonzero, the *arg* indicates that the talk transaction timed out. On most devices, some registers (usually register 0) generate a timeout if they are talked to but nothing is available to read. Other registers (for example, register 3) can always be talked to if the device exists, without a timeout occurring.

### Listening to the system

The fdb_listen routine instructs a device to "listen" to the system. You can call the fdb_listen routine as follows:

fdb_listen(*addr, id, register, datap, count*)

where:

- *addr* is the bus address of the device to be written (listened to).
- *id* is a number that is passed back to the driver with the service routine.
- *register* is the register being written (listened) to. This is a value between 0 and 3.
- *datap* is the address of a buffer that will contain the data being written.
- *count* is the number of bytes to be written.

After completion, the device's service routine is called with the *command* FDB_LISTEN. As in other routines, the *arg* indicates if a timeout has occurred.

## Polling

As noted earlier, the ADB hardware repeats the last talk transaction on the bus continuously if the bus is idle. If such a talk succeeds (that is, completes without timeout), then the processor is interrupted and the results of this talk are returned. Thus the device driver's service routine for that corresponding device will be called. In this case, the command is FDB_POLL and *arg* is the data returned from the successful talk.

Also, when a device with its service requests enabled (via a listen to the device) makes a service request, software must poll all known active devices. When the ADB software wants drivers to poll their respective devices, it calls the device service routines passing the command FDB_INT. The driver has the choice whether or not to initiate an ADB talk transaction to read from the device. If the driver chooses to, then it should return the value 1 from its service routine to indicate that a talk transaction has started. If for some reason the driver doesn't wish to start such a transaction (for example, it knows that it's device doesn't have service requests enabled, or that an ADB transaction is already in progress), then it returns 0.

If a service request is satisfied without polling all of the requested devices, then the service routines of those currently being polled are called with the command FDB_UNINT to indicate that their requests have been canceled.

Drivers can use their service routines to implement a Finite State Machine (FSM). This FSM would normally be started by a call to `fdb_open`. Such a FSM has two parts. The first part initializes the device. The second part consists of responses to device polls and is entered once the device is initialized.

Figure 11-1 shows the initialization states.

**Figure 11-1.**
**Initialization finite state machine diagram**

Once in the idle state, the device driver responds to the polling requests as shown in Figure 11-2.

**Figure 11-2.**
Polling finite state machine diagram

Note that the driver attempts to perform as many talks as possible until it receives a timeout. Thus, the hardware polls the device that performed the latest talk transaction, because a moved device is usually moved again soon (such as a mouse).

## A sample driver

The following is a sample skeleton interrupt routine for a device driver that implements the Finite State Machine just described. The comments marked with the string DEV should be replaced by the device prefix for your driver. Only the interface to the ADB driver is shown—the high-level interface could be to any type of A/UX device driver, such as a Streams or character device driver.

```
#define NDEVICES 1    /* the number of devices */
#define HANDLER       1       /* the device handler id */
static int DEV_state[NDEVICES];     /* current state */
static int DEV_present[NDEVICES];   /* TRUE if there really */
       /* is a DEV out there */
static short DEV_buff[NDEVICES];    /* where the fdb data is */
                                     /* read into */
static int DEV_intr();
#define STATE_INIT    0       /* not yet initialized */
#define STATE_IDLE    1       /* device is in inactive state */
#define STATE_REG3    2       /* register 3 listen in progress */
#define STATE_ACTIVE  3       /* register 0 talk in progress */
#define FDB_DEV       5       /* the fdb address of the device*/
/*
 *     called at spl7
 *            for each device
 *                   initialize its global variables
 *                          call fdb_open to declare the ISR and
 *                                 start the FSMs events
 */
```

```
DEV_init()
{
        register int i;
        for (i = 0; i < NDEVICES;i++) {
                DEV_state[i] =          STATE_INIT;
                DEV_present[i] =        0;
                fdb_open(FDB_DEV, i, DEV_intr);
        }
}
/*
 *      The device service routine
 */
static
DEV_intr(id, cmd, tim)
{
        switch(cmd) {
        case FDB_UNINT:
                /*
                 *      A poll was canceled .... mark the device as        inactive
                 */
                if (DEV_state[id] == STATE_ACTIVE)
                        DEV_state[id] = STATE_IDLE;
                break;
        case FDB_INT:
                /*
                 *      A poll is requested. If we are doing nothing
                 *      then do a fdb_talk to do the poll.
                 */
                if (DEV_state[id] == STATE_IDLE) {
                        fdb_talk(FDB_DEV, id, 0, &DEV_buff[id]);
```

```
                DEV_state[id] = STATE_ACTIVE;
                return(1);
        }
        return(0);
case FDB_POLL:
        /*
         *      A hardware poll succeeded ..... fake the
         *      timeout parameter and the DEV buffer to look
         *      as if a fdb_talk() succeded without timeout
         *      and fall through into the FDB_TALK handler
         */
        if (DEV_state[id] != STATE_IDLE &&
         DEV_state[id] != STATE_ACTIVE)
                break;
        DEV_buff[id] = tim;
        tim = 0;
case FDB_TALK:
        /*
         *      An ADB talk transaction completed.If it timed
         *      out mark the device as inactive and return. If
         *      it didn't pass the data read back to the user.
         *      If it wasn't a hardware poll start another
         *      transaction.
         */
        if (tim == 0) {              /* there is a message */

                /*
                 *      <- here pass the data back to the user
                 */
                if (cmd != FDB_POLL){
```

```
                fdb_talk(FDB_DEV, id, 0, &DEV_buff[id]);
                DEV_state[id] = STATE_ACTIVE;
            }
        } else {
            DEV_state[id] = STATE_IDLE;
        }
        break;
case FDB_LISTEN:
        /*
         *      The listen to set the handler id and service
         *      request enable has completed, now start a talk
         *      to register 0 to start the first device read
         *      transaction and to put the driver into
         *      the normal state.
         */
        DEV_state[id] = STATE_ACTIVE;
        fdb_talk(FDB_DEV, id, 0, &DEV_buff[id]);
        break;

case FDB_EXISTS:
        /*
         *      This is as a result from the fdb_open() in
         *      DEV_init() above. If tim is nonzero then the
         *      device does not exist. Tell any higher level
         *      drivers. If it does then start a flush
         *      transaction to clean out the device.
         */
        if (tim) {
            DEV_state[id] = STATE_INIT;
        } else {
```

```
                DEV_present[id] = 1;
                fdb_flush(FDB_DEV, id);
            }
            break;
        case FDB_FLUSH:
            /*
             *      After the flush completes start a listen to
             *      set the device's handler number and turn on
             *      the service request interrupts
             */
            DEV_state[id] = STATE_REG3;
            DEV_buff[id] = 0x2000 | (FDB_DEV<<8) | HANDLER;
            fdb_listen(FDB_DEV, id, 3, &DEV_buff[id], 2);
            break;
        case FDB_RESET:
            return;
        }
}
```

# Chapter 12

# Autoconfiguration

Autoconfiguration is an easy technique for adding, deleting, or replacing a device driver or software module in the A/UX kernel. Autoconfiguration involves three main programs: the `launch(8)` program, which loads the kernel into memory; the startup code of the kernel; and the `autoconfig(1M)` utility.

In addition, two other programs, `finstall` and `/etc/newunix`, are indirectly involved in the autoconfiguration process.Customers use `finstall` to initially install your software module onto their A/UX system, and `/etc/newunix` to prepare the files that `autoconfig` uses to link your driver into the kernel.

This chapter describes autoconfiguration with in-depth detail of the system activities that occur during the autoconfiguration process. This information is provided for completeness and to help you in adding your driver to the kernel.

In this chapter, you'll learn how to do the following:

- write your device driver using autoconfiguration guidelines
- learn what system activities happen prior to and during bootup that affect your driver
- create a master script file
- write optional initialization scripts to run after `autoconfig(1M)` links your driver into the kernel
- write optional startup scripts that run when the system is booted
- write an install script that is used with `/etc/newunix` to create the files that `autoconfig` uses to add your driver to the kernel
- run the `autoconfig(1M)` utility to add your driver to the kernel

This chapter outlines the main steps involved in adding a device driver to the A/UX kernel on the Macintosh II. Chapter 13 presents a specific example of using autoconfiguration in a driver development environment. After writing and successfully testing your driver, you should read Chapter 14 for details on how to prepare your driver and other files so that your customers can easily install your software.

## Introduction to the autoconfiguration process

When you turn on your A/UX system disk and turn on power to your Macintosh II computer, a number of activities occur "behind the scenes" before A/UX is actually booted. First, the Standalone Shell (SASH) application is executed. SASH then invokes the `launch` application.

`launch` loads the kernel into memory. Then `launch` probes the hardware and builds a data structure indicating which NuBus slots contain slot cards, recording the board id of each slot card. `launch` compares the current hardware configuration (of cards in NuBus slots) with the software configuration of the kernel.

If all software modules in the kernel that control slot cards have matching hardware, `launch` sets the AUTO_OK flag. If any software module that controls a slot card does not have the matching hardware present, or if the `-a` option is specified on the `launch` command line, `launch` sets the AUTO_RUN flag.

The value of AUTO_RUN or AUTO_OK is used later by `autoconfig` in determining whether a new kernel should be built.

After this initial processing, `launch` transfers execution to the kernel. The kernel begins the bootup process, executing the code in the pstart section of the kernel. Among other functions, the kernel begins setting up memory, and calls the *driver*`init` routines at various stages of the bootup process.

After this initial setup,the kernel executes the `init` process. The `init` process executes the lines in `/etc/inittab`, which includes a line that runs `/etc/sysinitrc`. Among other functions, `/etc/sysinitrc` executes `/etc/autoconfig`.

autoconfig is the utility that is responsible for automatically generating a new kernel when you add new hardware or drivers to the system.

The `autoconfig` utility is used in two ways. The kernel automatically executes `autoconfig` at boot time, to ensure that the software configuration of the kernel matches the hardware configuration in slot cards. You can also execute `autoconfig` from a running A/UX system, to generate a new kernel that you can boot later.

The `autoconfig` utility first determines if a new kernel should be built. If the AUTO_OK flag was set by `launch` and if `autoconfig` was invoked with the `-a` option, `autoconfig` does not build a new kernel, but immediately exits and the boot process continues.

If the AUTO_RUN flag was set by launch or if `autoconfig` was not invoked with the `-a` option, `autoconfig` proceeds to build a new kernel. After linking a new kernel, if `autoconfig` was invoked with the `-a` or `-I` option, `autoconfig` executes all driver initialization scripts found in the `/etc/init.d` directory.

After building a new kernel, if `autoconfig` was invoked with the `-a` option, `autoconfig` reboots the system. Rebooting the kernel will cause `autoconfig` to be invoked again. This time, the current hardware configuration matches the current software configuration, so `autoconfig` exits, and the boot process continues. If you have supplied a startup script for your driver, the kernel executes that script at this time.

After the system is booted and you see the `login:` prompt, you can log in and begin to use and test your driver.

## The files involved in the autoconfiguration process

A number of files are involved in the autoconfiguration process. The names and descriptions of the files related to the kernel that will be booted are as follows:

| | |
|---|---|
| `/newunix` | An A/UX kernel that contains only the minimum devices to boot an A/UX system. This is the original A/UX kernel shipped by Apple. |
| `/unix` | The currently running A/UX kernel or an A/UX kernel created by `autoconfig` to reflect customized changes to the kernel. By default, `autoconfig` builds the new kernel as `/unix`. |
| `/nextunix` | A file that contains the name of an A/UX kernel. This file originally contains the name `/unix`. |

The following is a list of programs involved in the autoconfiguration process:

| | |
|---|---|
| `launch` | A Macintosh application that resides on a small HFS partition on the A/UX system disk. The SASH application invokes `launch`, which probes the hardware for slot cards, loads an A/UX kernel into memory, and transfers execution to the kernel. |
| `/etc/autoconfig` | The program that builds a new kernel.The kernel executes `autoconfig` automatically at boot time.You can also execute `autoconfig` from a running A/UX system to build a new kernel that you can boot later. |
| `/etc/newunix` | The script that installs (or uninstalls) appropriate scripts and driver object files needed by `autoconfig`. The user executes this script to prepare to add new modules to the kernel. After executing `/etc/newunix`, the user should run `autoconfig` to create the new kernel. |

The following is a list of directories that `/etc/newunix` uses and the types of files stored in these directories:

| | |
|---|---|
| `/etc/install.d/*` | Installation scripts |
| `/etc/install.d/boot.d/*` | Driver object files |
| `/etc/uninstall.d/*` | Uninstallation scripts |

The following is a list of directories that `autoconfig` uses and the types of files stored in these directories:

| | |
|---|---|
| `/etc/master.d/*` | Master script files |
| `/etc/boot.d/*` | Driver object files |

| | |
|---|---|
| `/etc/init.d/*` | Device initialization scripts |
| `/etc/startup.d/*` | Startup scripts |

You need to supply certain information to `autoconfig` in order to add your driver to the kernel. This information is contained in files that you create and store in specific directories. These files and their contents are described in detail in the following sections.

The functions of `autoconfig` are illustrated in Figure 12-1 and are briefly described here. When building a new kernel, `autoconfig` uses `/newunix` to create the new kernel. Every software module that is to be added to the kernel must have a master script file in the `/etc/master.d` directory. The master script file of a module controls how that module will be linked into the kernel. The object file of the module must be located in the `/etc/boot.d` directory.

`autoconfig` processes the master script file for each module, links the modules into the kernel, and builds the new kernel in `/unix`. When `autoconfig` is run at boot time, `autoconfig` runs the programs in `/etc/init.d`, and creates the `/etc/startup` file. The `/etc/startup` file contains a list of the driver startup scripts that will be invoked at boot time.

Depending on various command line options that were specified to `autoconfig`, `autoconfig` may or may not reboot the kernel.

**Figure 12-1**
The functions of autoconfig

You must create the files required by `autoconfig` to add your driver to the kernel. After creating these files, you need to write an install script and uninstall script that can work with the `/etc/newunix` script. You should use the install script for your device to copy the object file, master script file, and other optional script files of your driver into the appropriate directories needed by `autoconfig`.

You must supply your users with the files required by `/etc/newunix`. You do this by putting these files on the same distribution disk as your driver. You can use `finstall` to copy the files from your distribution disk to specific directories of the A/UX system disk of your user. The `finstall` program is described in Chapter 14 of this manual.

The following section provides a quick reference guide to the steps involved in adding your driver to the kernel. Use this as a reference section only. Each step is explained in detail in later sections. Following the quick reference section,is a detailed explanation of `launch` and `autoconfig`. These sections will give you a deeper understanding of the bootup process. Following this discussion, specific directions to add your driver to the kernel are given.

The rest of this chapter uses the term *module* to describe a compiled object file suitable for linking with the kernel. Each module must have a companion master script file. The master script file is described in a following section.

The term driver is used to describe a piece of code that presents one of the A/UX block or character device interfaces to a user.

## Ten steps to add your driver to the kernel

This section provides a quick overview of the steps involved in adding your driver to the kernel. Refer to following sections which give specific information for each step. The following steps use the driver name *mydevice* to illustrate specific examples. To follow these steps for your device, replace the name *mydevice* with the name of your driver.

**1. Write your device driver.**

If you are writing a character device driver, your driver should contain the routines *mydevice*`open`, *mydevice*`close`, *mydevice*`read`, *mydevice*`write`, *mydevice*`ioctl`, and *mydevice*`select`, as appropriate for your device.

If you are writing a block device driver, your driver should contain the routines *mydevice*`open`, *mydevice*`close`, *mydevice*`strategy`, and *mydevice*`print`.

In addition, both block and character device drivers can provide a *mydevice*`init` routine, to perform initialization functions.

Device drivers can also provide an interrupt routine. For slot device drivers, you must name this interrupt routine *mydevice*`int`. Most other device drivers also follow this naming convention.

**2. Compile your device driver. Rename the object file and copy the object file to the `/etc/install.d/boot.d` directory.**

After compiling your driver, rename the resulting object file *mydevice.o* to *mydevice* (dropping the .o suffix). Copy this file to the `/etc/install.d/boot.d` directory.

Your install script (`/etc/install.d/`*mydevice*) invoked by `/etc/newunix` should copy your object file `/etc/install.d/boot.d/`*mydevice* to `/etc/boot.d/`*mydevice*. `autoconfig` looks in the `/etc/boot.d` directory for drivers or modules that need to be added to the kernel.

`/etc/newunix` installs or uninstalls the appropriate scripts and driver object files needed by `autoconfig`. `/etc/newunix` lets the user both determine the type of kernel to create and choose which of the available modules to include in the kernel.

**3. Create a master script file for your device.**

`autoconfig` uses information in the master script file to gain information on how to link your driver to the kernel. For example, the master script file tells `autoconfig` whether your driver is a block device driver, character device driver, streams driver, or streams module; whether your driver will receive interrupts from a slot card; and whether to create certain data structures (such as a `tty` structure) for your driver.

The master script file determines whether or not your driver gets included in the kernel. Your master script file must have an include statement or your driver must be included by another master script file to get included in the kernel.(See "Using module dependency information" for a description of the include statement.)

The master script file for your driver should be named *mydevice*. Your install script (`/etc/install.d/`*mydevice*) invoked by `/etc/newunix` should create your *mydevice* master script file and place it in the `/etc/master.d` directory.

Your install script can create a new file by using the `cat` or `echo` shell script command.

**4. Create an initialization script for your driver (optional).**

Initialization scripts are named *mydevice* and located in the `/etc/init.d` directory. These scripts are executed after `autoconfig` links your driver into the kernel, if `autoconfig` was invoked with the `-I` or `-a` option. Initialization scripts are typically used to create device files for your device.

Your install script (`/etc/install.d/`*mydevice*), which is invoked by `/etc/newunix`, should create your *mydevice* initialization script file and place it in the `/etc/init.d` directory.

**5. Create a startup script for your driver (optional).**

Startup scripts are named *mydevice* and located in the `/etc/startup.d` directory. Scripts in `/etc/startup.d` are executed every time the system is booted. You typically use startup scripts to create device files for your device or to download code to a controller.

Your install script (`/etc/install.d/`*mydevice*) invoked by `/etc/newunix` should create your *mydevice* startup script file and place it in the `/etc/startup.d` directory.

**6. Create an install script for your driver and place it in `/etc/install.d`. Also create an uninstall script and place it in `/etc/uninstall.d`.**

You should name the install script for your device *mydevice* and place this script in the `/etc/install.d` directory. You should name the uninstall script for your device no*mydevice* and place this script in the `/etc/uninstall.d` directory.

Scripts in `/etc/install.d` and `/etc/uninstall.d` are used with the `/etc/newunix` script. Your *mydevice* install script should copy your driver object file `/etc/install.d/boot.d/`*mydevice* to `/etc/boot.d/`*mydevice*, create the `/etc/master.d/`*mydevice* file, and create optional scripts in `/etc/init.d` and `/etc/startup.d` as needed for your device.

**7. Modify `/etc/newunix`**

Modify the `/etc/newunix` script so that it will accept the name of your driver as an argument, such as `/etc/newunix` *mydevice*. Also modify the script so that the user can specify `no`*mydevice* to uninstall your driver.

**8. Run `/etc/newunix mydevice`**

`/etc/newunix` will run the *mydevice* install script (located in `/etc/install.d`). Your install script should make sure that all files that `autoconfig` needs to include your driver into the kernel are placed in the appropriate directories.

Before running `autoconfig` or before rebooting the kernel, make sure you have backed up your currently running kernel. For example, execute the command `cp /unix /oldunix`.

**9. Run `autoconfig`**

Run `autoconfig` to create a new kernel. If you provided the necessary files and information to `autoconfig`, `autoconfig` will link your driver or module into the new kernel. You must specify the `-I` option to `autoconfig` if you have supplied an initialization script and want `autoconfig` to execute it. You must specify the `-S /etc/startup option` if you have a startup script that you want added to the list of startup scripts in `/etc/startup`.

If your hardware is already installed, then shutdown your system and reboot to begin running your new kernel.

If your hardware is not yet installed, powerdown your A/UX system and turn off power to all devices connected to your system. Install your hardware according to the instructions for your device. After installing your hardware, turn on all devices connected to your system. Turn on your computer; the system should begin the bootup process.

**10. Perform I/O to your device (test/debug)**

After you turn on your computer or reboot your system, the SASH application begins. SASH invokes `launch`, which loads the kernel into memory. `launch` checks the hardware configuration, sets either the AUTO_RUN or AUTO_OK flag, and then transfers execution to the kernel.

If your driver has a *mydevice*`init` routine, the kernel will invoke it during the bootup process, before the scheduler executes `init`. The `init` process is then scheduled, and `/etc/inittab` executes `/etc/sysinitrc`.

`/etc/sysinitrc` executes `autoconfig`. If `launch` set the AUTO_RUN flag and if the `-a` option was specified to `autoconfig`, `autoconfig` reboots the new kernel.

If you have installed your hardware and previously executed `autoconfig`, when you power on the computer or reboot the system, the kernel should not require updating and `autoconfig` will exit and the boot process continues. If you have supplied a startup script, the kernel will execute it during bootup.

The kernel finishes rebooting and you should see the `login:` prompt. You can now log in and perform I/O to your device.

## Background - the startup process

Autoconfiguration is a sequence of events that happens automatically at boot time. This section and the following sections explain the `launch` program, the sequence of events that occur at boot time that affect your driver, and the functions of the `autoconfig` utility.

The purpose of the autoconfiguration process is to check the consistency between the hardware attached to your Macintosh II in NuBus slots and the information in the kernel about software modules that control slot cards. If software configuration does not match the hardware configuration in slot cards, `autoconfig` automatically builds and reboots a new kernel.

The autoconfiguration process allows you to change your hardware configuration without changing DIP switches or manually rebuilding the kernel. For slot cards, autoconfiguration protects you against any problems caused by mismatched hardware and software.

## The launch program

`launch` is a Macintosh application that resides on a small HFS partition on the A/UX system disk. The three basic functions of `launch` are to load the kernel into memory, record which NuBus cards are installed in which NuBus slots, and transfer execution to the kernel.

The `launch` command accepts various options that can be specified in the command line. To examine or modify the `launch` command line, pull down the Preferences menu and then select Booting... . You can specify the command line options to `launch` by modifying the text in the box labeled Launch command: in the Booting...dialog box. Figure 12-2 shows the Booting...dialog box .

**Figure 12-2**
The launch command line

`launch` uses certain rules in determining which kernel to boot. If the `launch` command line specifies the name of the kernel, `launch` boots the specified kernel. For example, to boot a kernel called `/oldunix`, type `launch /oldunix` in the `launch` command line.

If `launch` is specified with the `-a` option, `launch` boots `newunix` as the kernel (for example, `launch -a`).

Otherwise, `launch` uses "the first line of `/nextunix`" as the kernel to boot. The first line of `/nextunix` usually contains `/unix`. In this case, `launch` boots `/unix`. If `/nextunix` doesn't exist, then `launch` boots `newunix`.

Note that `newunix`, when specified without a leading slash (/), has special meaning to `launch`. Whenever `launch` boots `newunix`, `launch` sets the AUTO_RUN flag, indicating that `autoconfig` should build a new kernel.

You can force `launch` to set the AUTO_OK flag by specifying the `-n` option on the `launch` command line. The `-n` option is useful when debugging a driver for a NuBus card that does not yet have the slot ROM installed.Refer to `launch(8)` in *A/UX System Administator's Reference* for more information on other command line options to `launch`.

After loading the specified kernel into memory, `launch` checks the consistency between the hardware attached to your Macintosh II in NuBus slots and the information in the kernel about software modules that control slot cards.

`launch` builds the `board_id` array and `version_id` array in the `auto_data` kernel data structure. For each NuBus slot, `launch` checks if a card exists in that slot. If so, `launch` reads the slot ROM to determine the board id and version id of the slot card, and stores this information in the `auto_data` structure.After cycling through the slots, the `auto_data` structure contains the board id and version number of each slot card that is present in a slot.

`launch` then determines if the hardware and software configuration matches by examining the `MODULES` section of the kernel. The `MODULES` section includes a data structure called `module` (defined in `<sys/module.h>`). For slot device drivers, the board id and version id fields of the `module` data structure contain the board id and version id of the slot card that the driver controls. The `module` structure also specifies what slot or slots the driver expects to find the card in.

For each `module` structure that specifies a board id, `launch` examines the `auto_data` structure, looking for a slot card with a matching board id. If `launch` does not find a slot card with a matching board id in an acceptable slot, `launch` sets the AUTO_RUN flag, which indicates that autoconfiguration should be run.

If all `module` structures that specify a board id have matching hardware (as indicated in the `auto_data` structure), `launch` sets the AUTO_OK flag, indicating that autoconfiguration does not need to take place.

The value of the AUTO_OK and AUTO_RUN flag is used later by `autoconfig` to determine whether a new kernel should be built.As previously described, `launch` sets AUTO_OK or AUTO_RUN according to whether the software configuration matches the hardware configuration. These settings can be overridden in the following situations:

- If the kernel name was specified as `newunix` in the `launch` command line, `launch` sets AUTO_RUN.

- If the `launch -n` option was specified in the `launch` command line, then `launch` sets the AUTO_OK flag. If the `-n` option is specified, `launch` sets AUTO_OK even if the hardware and software configuration doesn't match.
- If the `launch -a` option was specified in the launch command line, then `launch` sets the AUTO_RUN flag. If the `-a` option is specified, `launch` sets AUTO_RUN even if the hardware and software configuration does match.

## Booting the kernel

After `launch` finishes processing, `launch` transfers control to the kernel. The kernel begins the bootup process. This process includes setting up memory and calling *driver*`init` routines. After initial kernel processing, `/etc/init` (the "initial process") is executed.

`/etc/init` is the first A/UX process to run after booting the system. The `init` process runs before you enter single-user mode. `/etc/init` reads the lines in `/etc/inittab` and executes them.

The first command in `/etc/inittab` is the `/etc/sysinitrc` shell program. `/etc/sysinitrc` performs basic functions before you see the single-user mode shell prompt. For example, `/etc/sysinitrc` executes `/etc/autoconfig`, and then executes `/etc/startup`.

`/etc/sysinitrc` contains the following line, which executes `autoconfig`:

```
/etc/autoconfig -a -o /unix -S /etc/startup \
-M /etc/master #system configuration
```

The `-a` option to `autoconfig` means that `autoconfig` should build a new kernel only if `launch` has set the AUTO_RUN flag. Otherwise, `autoconfig` exits and the boot process continues.

If the `-a` option is not specified to `autoconfig`, or if the `-a` option is specified and the AUTO_RUN flag is set, `autoconfig` relinks a new kernel with `/newunix` and the object modules in `/etc/boot.d`. If the kernel was relinked and the `-a` option was specified to `autoconfig`, then `autoconfig` reboots the system. `/etc/sysinitrc` runs again and calls `autoconfig`. This time the kernel should be up to date, so `autoconfig` exits.

After `autoconfig` finishes execution, `/etc/sysinitrc` calls `/etc/startup`. `/etc/startup` runs the driver startup scripts for autoconfigured modules that are part of the kernel. The driver startup scripts are found in the `/etc/startup.d` directory.

## The autoconfig utility

`Autoconfig(1M)` is a utility that runs automatically at boot time and checks the consistency between the hardware that is attached to your Macintosh II in NuBus slots and the information in the kernel about slot cards.

Recall that `autoconfig` is used in two ways: at boot time, to automatically generate and reboot a new kernel (under certain conditions); and from a running A/UX system, to generate a new kernel that can later be booted.

The `autoconfig` utility accepts various command line options.Note that you must be superuser (root) to run the `autoconfig` program. Refer to `autoconfig(1M)` for a complete list of the command line options to `autoconfig` . The `-a` option is illustrated in Figure 12-3.

If you specify the `-a` option to `autoconfig`, `autoconfig` creates a new kernel only if `launch` has set the AUTO_RUN flag. If you specify the `-a` option and `launch` has set the AUTO_OK flag, `autoconfig` exits and the boot process continues.

If you do not specify the `-a` option to `autoconfig`, `autoconfig` proceeds with the entire configuration process and creates a new kernel, regardless of the value of the AUTO_OK or AUTO_RUN flag.

If `autoconfig` created a new kernel and if the `-a` option was specified, `autoconfig` reboots the kernel.

Note that `autoconfig` is invoked with the `-a` option from the `/etc/sysinitrc` script. When you use `autoconfig` during driver development, you usually will not specify the `-a` option. By not specifying the `-a` option, you can add other software modules to a new kernel that you can later boot from.

**Figure 12-3**
An overview of autoconfig

`autoconfig` begins to create a new kernel by making a list of the present modules in the kernel. `autoconfig` then searches the `/etc/master.d` directory for master script files to process. `autoconfig` processes each master script file. If the module is not already in the kernel, `autoconfig` adds the module to a list of possible modules to be included.

`autoconfig` creates a `module` data structure to describe each module to be included in the kernel. `autoconfig` fills in the `module` structure with information contained in the master script files. For example, `autoconfig` records if the module is a block device driver or character device driver, a streams driver or module, whether the module receives interrupts, and when the *driver*`init` routine should be called.

For each module represented by a master script file, `autoconfig` checks if the master script file defines a board id and version number. If the master script file defines a board id and version number, `autoconfig` records this information in the `module` data structure.

`autoconfig` processes the master script file for any dependency statements. Each module is marked included or excluded from the kernel according to the evaluation of the dependency statements.If the master script file includes a dependency statement that specifies that a particular module be included, `autoconfig` looks for the object module in `/etc/boot.d` to include in the final link of the kernel.

Next `autoconfig` processes the last line of each master script file. This line contains the flags, number of interrupt vectors, driver prefix, major number, number of devices, and interrupt priority level for the module. `autoconfig` records this information and takes various actions depending on the values specified on this line.

Then `autoconfig` verifies that each slot card has a module that controls it. For each card, `autoconfig` gets the board id and version number of the card from either the `auto_data` structure or the loadfile. `autoconfig` then searches the modules for a module with a matching board id and version number.

Any module that has a corresponding slot card with a matching board id and version number is marked as to be included in the kernel.

For a slot device driver, your master script file must tell `autoconfig` the board id of the slot card your device driver controls. Doing this allows `autoconfig` to check if the slot card really exists in the system. If the slot card is installed and the board id of the slot card matches the declared board id of a module, `autoconfig` will link the driver into the kernel. If the slot card is not present, `autoconfig` will not link the driver into the kernel.

For any slot card that `autoconfig` cannot find a module with a matching board id and version number, `autoconfig` prints a warning message to the system console. For example, if `autoconfig` cannot find the driver for the EtherTalk card, `autoconfig` prints a message similar to the following:

```
Warning cannot find driver(s) for device ID 5
Version 7.0
```

After processing the master script files, `autoconfig` prepares the new kernel for linking. This preparation includes allocating major numbers to new modules and setting up various kernel data structures, such as the cdevsw and bdevsw tables.

`autoconfig` also sets up the slot interrupt vector table. For slot device drivers, `autoconfig` sets up the slot interrupt vector table entry for your card to contain the address of your interrupt routine. The interrupt routine of slot device drivers must be named *driver*`int`, where *driver* is the device prefix specified in your master script file. Naming your interrupt routine *driver*`int` allows `autoconfig` to set up the appropriate entry in the slot interrupt vector table to contain the address of your *driver*`int` routine.

`autoconfig` runs `/bin/ld` to link the new modules and `/newunix` into the new kernel. The new kernel is created as `/unix`, unless otherwise specified in the `autoconfig` command line.

If `autoconfig` was invoked with the -S *file* option, `autoconfig` makes a list of the modules that have startup scripts or programs which the kernel is to call at boot time. `autoconfig` puts this list of startup programs to call in the file specified on the command line (usually `/etc/startup`). When the kernel boots, the startup scripts of the modules listed in `/etc/startup` are executed.

If `autoconfig` was invoked with the -I option, then `autoconfig` executes the driver init scripts found in `/etc/init.d` that correspond to modules in the new kernel.

`autoconfig` does final processing, including the writing of a summary of the autoconfiguration results to the system console. If `autoconfig` was invoked with the `-a` option, `autoconfig` reboots the kernel.

## The /etc/newunix script

For `autoconfig` to include your driver into the kernel, you need to provide a master script file and the object file of your driver. You can also optionally provide a startup script and an init script for your driver. These files must be located in the following directories:

```
/etc/master.d

/etc/boot.d

/etc/init.d

/etc/startup.d
```

Once a master script file is placed in `/etc/master.d` with a companion object file in `/etc/boot.d`, the next time `autoconfig` is run (without the `-a` option), `autoconfig` will create a new kernel, including the new module in the kernel.

Rather than directly place these files in `/etc/master.d` and `/etc/boot.d`, you should let the user explicitly place the files in these directories by using `/etc/newunix`. You do this by writing an install script that can be invoked by `/etc/newunix`. The install script for your module should be located in `/etc/install.d`. Your install script should set up the files `autoconfig` needs to include your driver into the kernel.

The user specifies one or more arguments to `/etc/newunix`. Each argument corresponds to one or more modules that are to be included in the new kernel.

`/etc/newunix` lets the user determine the type of kernel to create and choose which of the available modules to include in the kernel. The user uses `/etc/newunix` to begin the process of configuring a new kernel. For each argument specified to `/etc/newunix`, `/etc/newunix` executes an install or uninstall script for that argument. The install script for a particular argument installs the scripts and driver object files needed by `autoconfig` to configure that module into the kernel. The uninstall script for a particular argument removes the files used by `autoconfig` for that module.

The arguments specified to `/etc/newunix` depends on the type of kernel you want to create: basic networking (`bnet`), Network File System (`nfs`), A/UX toolbox (`toolbox`), non-networking (`nonet`), no toolbox capabilities (`notoolbox`).

You must modify `/etc/newunix` to include processing of your install and uninstall script.

You should backup your current `/etc/newunix` file, then edit `/etc/newunix`. Add a case statement for the name of your driver. Inside the case statement add a line that executes your install script. Also add a case statement that will execute your uninstall script.

You should either provide this modified version of `/etc/newunix` to your customers on your distribution disk, or include directions for your customers to edit `/etc/newunix` so that they can make the changes themselves.

Install scripts usually copy the file in `/etc/install.d/boot.d/`*mydevice* to `/etc/boot.d/`*mydevice*. Most install scripts create the other files (the master script file, startup scripts, and init scripts) by creating the file in-line by using either the `echo` or `cat` shell command.

The install script that you write for `/etc/newunix` should be called *mydevice*, where *mydevice* is the device prefix of your driver. Place this script in the `/etc/install.d` directory. Your install script should install the necessary files in the `/etc/master.d`, `/etc/boot.d`, `/etc/init.d`, and `/etc/startup.d` directories.

The uninstall script that you write for `/etc/newunix` should be called `no`*mydevice*, where *mydevice* is the device prefix of your driver. Place this script in the `/etc/uninstall.d` directory. Your uninstall script should remove the necessary files in the `/etc/master.d`, `/etc/boot.d`, `/etc/init.d`, and `/etc/startup.d` directories.

## The driver development process

Figure 12-4 shows the stages in developing and installing a device driver using autoconfiguration.

Once a master script file is placed in `/etc/master.d` with a companion object file in `/etc/boot.d`, the next time `autoconfig` is run (without the `-a` option), `autoconfig` will create a new kernel, including the new module in the kernel.

Rather than directly place these files in `/etc/master.d` and `/etc/boot.d`, you should let the user explicitly place the files in these directories by using `/etc/newunix`. You do this by writing an install script that can be invoked by `/etc/newunix`. The install script for your module should be located in `/etc/install.d`. Your install script should set up the files `autoconfig` needs to include your driver into the kernel.

The user specifies one or more arguments to `/etc/newunix`. Each argument corresponds to one or more modules that are to be included in the new kernel.

`/etc/newunix` lets the user determine the type of kernel to create and choose which of the available modules to include in the kernel. The user uses `/etc/newunix` to begin the process of configuring a new kernel. For each argument specified to `/etc/newunix`, `/etc/newunix` executes an install or uninstall script for that argument. The install script for a particular argument installs the scripts and driver object files needed by `autoconfig` to configure that module into the kernel. The uninstall script for a particular argument removes the files used by `autoconfig` for that module.

The arguments specified to `/etc/newunix` depends on the type of kernel you want to create: basic networking (`bnet`), Network File System (`nfs`), A/UX toolbox (`toolbox`), non-networking (`nonet`), no toolbox capabilities (`notoolbox`).

You must modify `/etc/newunix` to include processing of your install and uninstall script.

You should backup your current `/etc/newunix` file, then edit `/etc/newunix`. Add a case statement for the name of your driver. Inside the case statement add a line that executes your install script. Also add a case statement that will execute your uninstall script.

You should either provide this modified version of `/etc/newunix` to your customers on your distribution disk, or include directions for your customers to edit `/etc/newunix` so that they can make the changes themselves.

Install scripts usually copy the file in `/etc/install.d/boot.d/`*mydevice* to `/etc/boot.d/`*mydevice*. Most install scripts create the other files (the master script file, startup scripts, and init scripts) by creating the file in-line by using either the `echo` or `cat` shell command.

The install script that you write for `/etc/newunix` should be called *mydevice*, where *mydevice* is the device prefix of your driver. Place this script in the `/etc/install.d` directory. Your install script should install the necessary files in the `/etc/master.d`, `/etc/boot.d`, `/etc/init.d`, and `/etc/startup.d` directories.

The uninstall script that you write for `/etc/newunix` should be called `no`*mydevice*, where *mydevice* is the device prefix of your driver. Place this script in the `/etc/uninstall.d` directory. Your uninstall script should remove the necessary files in the `/etc/master.d`, `/etc/boot.d`, `/etc/init.d`, and `/etc/startup.d` directories.

## The driver development process

Figure 12-4 shows the stages in developing and installing a device driver using autoconfiguration.

**Figure 12-4**
Developing and Installing a device driver

The following sections describe each step of the driver development process in detail.

## Writing and compiling your device driver

Chapters 2 through 11 showed how to write a device driver. Refer to these chapters for detailed information about writing your driver.

When you write your device driver, you should follow certain naming conventions. Remember that you should give your device driver routines a unique three to eight character prefix that is affixed to a routine name. Valid characters are alphanumeric or an underline (_). For example, if you use the prefix `MYDEVICE`, you should name the open routine for your driver `MYDEVICEopen`.

Remember that since your driver will coexist with many other drivers, you should declare any data structures and routines that are not referenced outside of your driver as static.

Table 12-1 shows a list of the names of routines for character device drivers. Your character device driver must follow these naming conventions.

Table 12-2 shows a list of the names of routines for block device drivers. Your block device driver must follow these naming conventions.

**Table 12-1**
Routine naming conventions for character device drivers

| Routine name | Description |
|---|---|
| *prefix*`open` | Character device open routine |
| *prefix*`close` | Character device close routine |
| *prefix*`read` | Character device read routine |
| *prefix*`write` | Character device write routine |
| *prefix*`ioctl` | Character device ioctl routine |
| *prefix*`select` | Character device select routine |
| *prefix*`info` | Stream device interface structure |
| *prefix*`tty` | tty structure for terminal device drivers |
| *prefix*`init` | Device initialization routine (optional) |
| *prefix*`int` | Device interrupt routine(optional) |
| *prefix*`driver` | B-NET network interface |
| *prefix*`fork` | fork execution routine |

| | |
|---|---|
| *prefix*`exec` | exec execution routine |
| *prefix*`exit` | exit execution routine |

**Table 12-2**
Routine naming conventions for block device drivers

| Routine name | Description |
|---|---|
| *prefix*`open` | Block device open routine |
| *prefix*`close` | Block device close routine |
| *prefix*`strategy` | Block device strategy routine |
| *prefix*`print` | Block device print routine |
| *prefix*`init` | Device initialization routine (optional) |
| *prefix*`int` | Device interrupt routine(optional) |
| *prefix*`driver` | B-NET network interface |
| *prefix*`fork` | fork execution routine |
| *prefix*`exec` | exec execution routine |
| *prefix*`exit` | exit execution routine |

After writing your driver, compile it using the `-c` flag to produce an object module. Rename this module giving it a name that uniquely identifies it, but without a `.o` filename extension. Then, move the renamed module to the `/etc/install.d/boot.d` directory.

## Creating the master script file

After you write your device driver, you must prepare a **master script file**. A master script file contains information used during autoconfiguration. Your install script (`/etc/install.d/`*mydevice*) should create the master script file for your device in the `/etc/master.d` directory.

The master script file for your driver can define driver characteristics, assign an identifying number to a slot card, or set up dependencies between modules. Give this file the same name that you chose for your renamed object module. A master script file can have three parts, as shown in Figure 12-5.

**Figure 12-5**
The master script file

A sample master script file for the EtherTalk card, found in the file `/etc/master.d/ae6` is shown below:

```
id 8
if . include slots
anvs    1     ae6    -      1      1
```

Of the information in this file, the first line contains a device identifier, which shows a board id of 8 for the EtherTalk card. The second line presents module dependency information. This information indicates that if the object file for the current module (ae6) exists, then `autoconfig` should include the `slots` module in the kernel. The third line shows device information, which specifies,among other things,that the device uses a network interface.

The following sections explain the information that you can specify in a master script file.

## Using a device identifier with slot devices

The **device identifier** is optional information that is specified only for slot device drivers. The device identifier is used to specify a particular slot card. The device identifier has the following syntax:

`id` *name serial*

where:

*name* is an integer board ID, which is stored in the slot ROM. This number must have been obtained from Apple Technical Support.

*serial* is optional information. If you use this field, `autoconfig` compares the number you specify here with the slot card's version number. If the two numbers do not match, `autoconfig` will not include the module in the kernel. If you do not fill in a value for this field, `autoconfig` will not check the slot card's serial number for a match. In this case, if the board id's match, `autoconfig` will include the module in the kernel.

*serial* can be one of the following:

| | |
|---|---|
| *number* | The slot card's serial number. `autoconfig` checks for an exact match between *number* and the version number. |
| *number*- | A serial number greater or equal to *number*. `autoconfig` checks if *number* is greater than or equal to the version number. |

| | |
|---|---|
| - *number* | A serial number less than or equal to *number*. `autoconfig` checks if *number* is less than or equal to the version number. |
| *number1* - *number2* | A serial number greater than or equal to *number1* and less than or equal to *number2*. `autoconfig` checks the version number for a value in this range. |
| empty | If you do not specify the number, `autoconfig` does not check the version number. |

`autoconfig` searches a slot card's revision level string and attempts to create a version number from it. `autoconfig` first looks for substrings having the forms *n.n* or *n*, where *n* can be one or two decimal numbers. Then `autoconfig` creates a new number by placing the first *n* in the hundreds place value and by appending the remaining digits to it (or zeros if no digits appear after the decimal point). For example,

3.01 becomes 301

3.1 becomes 301

31 becomes 3100

3.23 becomes 323

`autoconfig` then compares this new integer with the version range from the master script file. If `autoconfig` does not find a number in the string,`autoconfig` assumes 0.

## Using module dependency information

Depending on your needs, your master script file can contain include statements and other dependency statements.

When dependency processing begins, `autoconfig` reads and marks modules that are currently in `/newunix` (such as, the console driver and the root file system driver). These modules are marked as included.

Next, `autoconfig` marks every module with a master script file that has a device identifier name and a version number that matches a card in the current hardware configuration as to be included.

Next, `autoconfig` scans all driver master script files for any dependency statements. Each module is marked included or excluded according to the evaluation of the dependency statements.

If your driver is not a slot device driver and if your driver does not depend on any other modules, or is not included by another module, you *must* include the statement `"include ."` in your master script file to include your driver in the kernel.

Dependency statements have either this form

*verb namelist*

or this form

`if` *`ifexpr verb namelist`*

where

*verb* is either `include` or `exclude`. The term `include` tells `autoconfig` to include the modules in *namelist* in the kernel. The term `exclude` tells `autoconfig` to exclude the modules in *namelist* from the kernel.

*ifexpr* is a filename (use a period [.] for the current module) or an expression. If the filename exists or if the evaluation of the expression is true, then the modules specified in *namelist* are included in or excluded from the kernel. The expression can be negated (! ), AND'ed (&) or OR'ed ( | ) to another expression. The symbol ! has the highest priority, followed by &, and |. For example

a | b&c means a | (b&c) not (a | b)&c

!a&b means (!a)&b not !(a&b)

To override this priority, use parentheses.

*namelist* can be one or more module names (or a period [.] for the current module) separated by commas. `autoconfig` scans the master script files, evaluates the ifdef statements , and adds other modules if necessary.

Avoid circular dependencies. For example, suppose modules A, B, and C contain the following dependency statements:

```
A       if . include B

B       if . include C

C       if . include A
```

Neither A, B, nor C will be included. Note that the order in which the include statements appear does not matter. Also watch out for dependencies that contradict each other. For example,

```
A       include C

B       exclude C
```

will include and then exclude C. Both of these types of errors generate error messages and terminate autoconfiguration.

## Including device information

The device information is a required information line that tells `autoconfig` how to place the device driver into the kernel. This line contains the following six fields:

*flags nvec prefix soft devices ipl*

where

*flags* can contain a number of values.These values are described in detail later in this section.

*nvec* is the number of interrupt vectors that a particular controller can generate. For drivers that receive slot interrupts this number is 1 (because each controller can generate only one interrupt). For software modules that do not directly receive interrupts, this value should be a hyphen (-).

*prefix* is the prefix used in the driver's interface routines. The prefix must be between three and eight characters long. Valid characters are alphanumeric characters or an underline (_).

*soft* is a value used to assign the major number to your software driver. This value should always be a hyphen (-).When you specify a hyphen in this field, `autoconfig` assigns the first available major number not already assigned in the kernel to your device. Doing this prevents your major number from being used by any other device driver in that A/UX kernel. Doing this allows flexibility and guarantees that your driver is assigned a unique major number.

To create a device file for your driver, you need to know the major number for your device. Both the driver init scripts and driver startup scripts are passed a parameter that indicates the major number of your device. You should use the major number to create your device file at this time. Driver initialization scripts and driver startup scripts are explained in the section "Writing Optional Init and Startup Scripts".

*devices* is either a hyphen (-) for modules that aren't device drivers, or a nonzero integer for device drivers. The integer value is the number of devices that the controller supports. For example, if the controller supports only one device, this value should be 1. If the controller supports 8 devices, this value should be 8. This value is usually used with the *a* flag (described in a following paragraph).

*ipl* is the highest priority interrupt level used by the controller. For modules that don't receive interrupts, this value should be a hyphen (-). For slot-based devices (all of which interrupt at `spl1`), this value should be 1.

*flags* can be one or more of the following:

a This flag is used to create various data structures to be included in the kernel. In addition, you can use this flag with other flags to create data structures specific to terminal and block device drivers.

When the flag a is specified, two data structures are created:

*prefix*`cnt`

*prefix*`addr`

These data structures will be created and included in the final linked kernel. These data structures contain hardware configuration information and should be declared as `extern` in your driver.

*prefix*`cnt` is initialized with the number of controllers associated with your device driver in the system (not the number of devices). For example, assume your device driver controls video cards, and there are two video cards, a coprocessor card, and a networking card in the system. Then *prefix*`cnt` will have a value of 2 for your driver.

*prefix*`addr` is an array (having *prefix*`cnt` entries) containing the address of each controller. For slot device drivers, this value is the number of the slot that the slot card is in. You can use this value to map slot numbers to logical units or instances of your driver. For example, assume your device driver controls video cards, and there are currently two video cards in the system. These two video cards are in slots 9 and 12. Then *prefix*`cnt` equals 2, and *prefix*`addr` contains the following values:

*prefix*`addr`[0] = 9

*prefix*`addr`[1] = 12

You can use the values in *prefix*`addr` to calculate the base of the slot space for each card installed in the given slot.

For ADB and SCSI devices, *prefix*`addr` contains the ADB or SCSI address.

If you specify both the a and the t flag, an uninitialized array of type `struct tty` named *prefix*`tty` is created with (*prefix*`cnt` * *prefix*`addr`) number of entries.

If you specify both the a and the b flag, two arrays having the same dimensions are declared:

```
struct iobuf prefixiobuf[];
struct iostat prefixiostat[];
```

The number of elements in the *prefix*`iobuf` and *prefix*`iostat` arrays for your driver is equal to the number of devices times the number of controllers.

b This flag is used to specify a block device driver. Entry points to the driver will be added to the block device switch table. `autoconfig` looks for the routines with the names *prefix*`open`, *prefix*`close`, *prefix*`strategy`, and *prefix*`print`. For each of these routines `autoconfig` finds, `autoconfig` fills in the corresponding bdevsw entry with a pointer to the routine. If `autoconfig` does not find an entry for a bdevsw routine, the corresponding bdevsw entry will contain a default entry. The default entry is `nodev` or `nulldev`.

c This flag is used to specify a character device driver. Entry points to the driver will be added to the character device switch table. `autoconfig` looks for the routines with the names *prefix*`open`, *prefix*`close`, *prefix*`read`, *prefix*`write`, *prefix*`ioctl` and *prefix*`select`. For each of these routines `autoconfig` finds, `autoconfig` fills in the corresponding cdevsw entry with a pointer to the routine. If `autoconfig` does not find an entry for a cdevsw routine, the corresponding cdevsw entry will contain a default entry. For example, if you do not supply a *driver*`select` routine, `autoconfig` fills in the *driver*`select` entry in the cdevsw table for your device with the default entry `seltrue`. For most other routines, the default entry is `nodev` or `nulldev`.

l This flag is used to indicate that the module contains the line discipline code. `autoconfig` looks for routines named *prefix*`open`, *prefix*`close`, *prefix*`read`, *prefix*`write`, *prefix*`ioctl`, *prefix*`input`, and *prefix*`output` and fills in the line discipline switch entry with a pointer to the corresponding routine. You can only use the p flag with the l flag.

m This flag is used to indicate a Streams driver. `autoconfig` looks for a structure named *prefix*`info` and fills in the corresponding entry in the cdevsw table with a pointer to this structure.

n This flag is used to indicate that this device uses a network interface (TCP/IP).

p*opt* This flag lets you specify when a driver's initialization routine is called. All device drivers can supply a routine named *prefix*`init`. `autoconfig` will look for this routine in your driver. If `autoconfig` finds a routine named *prefix*`init`, `autoconfig` records that your *prefix*`init` routine should be called during bootup.

If you want your *prefix*`init` routine to be called at a particular time during system initialization and if you want to specify whether interrupts should be enabled or disabled, you can use the p flag with one of the following values for *opt*:

f Call *prefix*init routine first, before any other initialization occurs. Interrupts are disabled.

s Call *prefix*init routine second, after any `pf` modules are initialized. Interrupts are disabled.

n Call *prefix*init routine normally, after pf and `ps`, but prior to enabling interrupts. This is the default if you do not specify any p *opt* flag.

0 Call *prefix*init routine after interrupts are enabled.

1 Call *prefix*init routine after system starts but before the kernel enters `/etc/init`.

If your driver has a *prefix*`init` routine and you do not specify the p *opt* flag, your *prefix*`init` routine will be called as if you specified p n (normal).

`s` This flag is used to indicate a software module that is not linked to the system through the driver interface. It is used for modules such as subroutine libraries. You can only use the `p` flag with the `s` flag.

`t` This flag is used to indicate a character device driver that has a `tty` structure associated with it. `autoconfig` creates a global pointer to the `tty` data structure. You must use the `t` flag in conjunction with the `c` flag. Each `tty` structure is named *prefix*`tty` and is indexed by using the device's minor number.

`v`*opt* This flag instructs `autoconfig` to link your driver to the interrupt vector mechanism. Currently, the only value of *opt* supported is `s`, which indicates that the kernel is to decode slot-based interrupts and to call the interrupt routine of this driver when the card generates an interrupt.

If you specify `vs`, `autoconfig` adds your *prefix*`int` routine to the slot interrupt vector table.

`x` This flag is used to specify a Streams module. You can only use the `p` flag with the `x` flag.

`L` This flag specifies that this module is a COFF library.

`S`*opt* This flag is used to define special applications-defined exit, fork, and exec routines. Values of *opt* are as follows:

- `e` The module contains a routine *prefix*`exit`, which is called whenever a process exits.
- `f` The module contains the routine *prefix*`fork`, which is called whenever a process forks.
- `x` The module contains the routine *prefix*exec, which is called whenever a process execs a new image.

## Sample master script files

The following section shows four master script files for a device called `MYDEVICE`: the first one is for a character driver, the second one is for a block driver, and the last two are for a Streams driver and module.

## A character device driver master script file

Figure 12-6 presents an example of a sample character device driver master script file. In the example, the first line indicates that this driver controls a slot card with board id 99. The second line indicates that this driver depends on the slots module, and instructs `autoconfig` to include the slots module in the kernel if the `MYDEVICE` module is also included in the kernel.

In the third line, this script identifies a character terminal device (`ct`) whose interrupts are slot based (`vs`). Autoconfiguration will create two data structures for this module: `MYDEVICEcnt`, which contains the number of slot cards with board id 99 in the system, and `MYDEVICEaddr`, which is an array initialized with the slot number of each slot card controlled by the driver.

The 1 indicates that the module will receive one interrupt per controller (which is true for all slot devices). The rest of the line indicates that the device prefix is `MYDEVICE`, the software major number will be assigned by `autoconfig (-)`, there are 8 devices per controller, and the device *ipl* (the interrupt level at which the device takes interrupts) is `1`.

**Figure 12-6**

A sample master script file for a character device driver

## A block device driver master script file

A sample master script file for a block device driver is illustrated below and explained in the following paragraphs.

```
if . include SCSI
bca     -       MYDEVICE        -       2       1
```

In this script, the `if . include SCSI` statement on the first line assumes that another module (the SCSI manager) must be in the kernel for the module to run. In the second line, the `bc` indicates that the driver will be used as both a block and character device driver. `autoconfig` will create entries in both the bdevsw and cdevsw table for this device driver. This device driver shares open and close routines between the two device drivers. The `a` flag instructs `autoconfig` to create the `MYDEVICEcnt` and `MYDEVICEaddr` data structures.

The first hyphen (-) indicates that this module does not receive interrupts directly, because this device receives interrupts via the SCSI manager. The device's prefix is `MYDEVICE`, and `autoconfig` will assign the software major number `(-)`. There are 2 devices per controller and the device interrupt *ipl* is 1.

## A streams driver master script file

A sample master script file for a streams device driver is illustrated below and explained in the following paragraphs.

```
if . include STREAMS
mvsa    1       MYDEVICE        -       2       1
```

The first line of the script includes the STREAMS module into the kernel.

The second line of the script identifies a streams device (`m`) whose interrupts are slot based (`vs`). autoconfig will create the `MYDEVICEcnt` and `MYDEVICEaddr` data structures (`a`). The number of interrupt vectors the controller can generate is 1 (because it is slot based), the device prefix is `MYDEVICE`, and the software major number is not used. There are 2 devices per controller and the device *ipl* is 1.

## A streams module master script file

A sample master script file for a streams module is illustrated below and explained in the following paragraphs.

```
include .
x       -       MYDEVICE        -       -       -
```

Because this module doesn't depend upon any other module in the system, it must be explicitly included with the `include` . statement. This script first identifies a streams file (x). The script then specifies that no interrupt vectors are received (-), `MYDEVICE` is the device prefix, the software major number is not assigned (because this is a streams module), there are no devices per controller (because this module isn't a physical device), and the *tpl* is not applicable.

## Writing optional init and startup scripts

You might choose to write two optional types of scripts: device initialization scripts that run immediately after autoconfiguration, and startup scripts that run whenever the system boots or reboots an autoconfigured kernel.

Your install script (`/etc/install.d/`***mydevice***) should create the init and startup scripts for your device. Your install script should place your init script in the `/etc/init.d` directory, and your startup script in the `/etc/startup.d` directory.

If the `-I` or `-a` option is specified to `autoconfig` , `autoconfig` executes any files in `/etc/init.d` after building a new kernel but prior to rebooting.

Usually, these scripts create device files in the system's `/dev` directory. Naming conventions for device files are listed in the following section. A number of special programs such as `dev_kill(1M)`, `tty_kill(1M)`, and `tty_add(1M)` can be run. See *A/UX System Administrator's Reference* for details about these programs.

When you write init scripts, be careful about writing an init script that modifies the currently running environment while running the old kernel. You should place functions that could affect the currently running environment in your startup script.

Any files in `/etc/startup.d` whose names correspond to modules in the kernel are executed from `/etc/sysinitrc` (by the `/etc/startup script`) before entering single-user mode.

A list of these optional startup scripts is kept in the `/etc/startup` file, which is generated during autoconfiguration. When you run `autoconfig` , you must specify the `-S /etc/startup` option if you have a startup script that you want added to the list of startup scripts in `/etc/startup`. Usually, these startup scripts or programs perform initialization functions, such as downloading code to an intelligent controller.

The first flag passed to a device initialization script is `-d` and the first flag passed to a device startup script is `-s`. These flag options are passed in the following order to all startup and initialization scripts:

`-M` ***n*** The major number of this device type is ***n***. Only block, character, and streams drivers are passed the major number flag.

`-C` ***n*** There are ***n*** controllers associated with this module in the system.

`-D` *n* There are *n* devices per controller associated with this module in the system.

`-S` *n* There is a controller for this device type in NuBus slot number *n*. *n* is a single hex digit from 0 to 0x0F. See the *v* option in "Including Device Information" given earlier in this chapter for information about specifying slot interrupts. The `-S` flag is passed only if at least one controller for your device is actually installed in the system.

If more than one slot card for your device driver is installed in the system, then your script will be passed more than one `-S` flag. where the number following each `-S` is the slot number of one of your slot cards.

For example, the startup script of a slot device driver might be invoked as follows:

```
/etc/startup.d/TEST -M 9 -C 1 -D 1 -S 11
```

This line indicates that the `TEST` module has major number 9, one slot card associated with it in the system, one device per slot card, and is installed in slot 11.

## Device file naming conventions

You must create one or more device files in order to perform I/O to your device. Device files are typically created in the init or startup scripts of a driver. Recall that the init and startup scripts are passed the major number of the device. You can use this information to create your device files in these scripts.

This section uses the following terms:

card — A card supporting one or more units (usually, but not always) of the same device type. A card is also often referred to as a controller.

unit — A single physical device that can be individually addressed. For example, a unit could be one channel on a dual channel serial chip or one disk driver on a controller.

The device names in `/dev` and `/etc/inittab` should follow these naming conventions:

| Type | Name | `/etc/inittab` label |
|---|---|---|
| **Terminal devices** | `/dev/tty`*su* | tty*SU* |
| **Disks** | `/dev/dsk/c`*n*`d`*m*`s`*y*<br>`/dev/rdsk/c`*n*`d`*m*`s`*y* | |

where *n* is the SCSI ID of the Hard Disk SC, *m* is the number of the sub-driver at that SCSI ID (usually 0), and *y* is the slice number associated with a particular partition.

| | | |
|---|---|---|
| **Printers** | `/dev/lp`*su* | p*axu* |
| | `/dev/lp`*axu* | p*axu* |
| **Other devices** | `/dev/`*nameu* | *nameu* |
| | `/dev/`*name/xxx* | *nameu* |

where

*s* is the slot number.

*u* is the unit number.

*a* is other bus type (either S for local SCSI bus or F for Apple DeskTop Bus).

*x* is other bus index.

*name* is the driver name.

*xxx* is any letter or digit.

*Note:* You can place special files in `/dev` subdirectories to make searching through the directory faster.

## Creating the install and uninstall scripts

Once your driver, master script, and optional scripts are complete, you should write an install script for your device driver. This script should install the device driver, master script, and other files into the appropriate directories for `autoconfig`.

The install script for your device should be named *mydevice* and placed in the `/etc/install.d` directory. The uninstall script for your device should be named no*mydevice* and placed in the `/etc/uninstall.d` directory.

Scripts in `/etc/install.d` and `/etc/uninstall.d` are used with the `/etc/newunix` script. Your *mydevice* install script should copy your driver object file `/etc/install.d/boot.d/`*mydevice* to `/etc/boot.d/`*mydevice*, create the `/etc/master.d/`*mydevice* file, and create optional scripts in `/etc/init.d` and `/etc/startup.d` as needed for your device.

Your uninstall script no*mydevice* should remove the files related to *mydevice* in the `/etc/master.d`, `/etc/boot.d`, `/etc/init.d`, and `/etc/startup.d` directories.

## Modifying /etc/newunix

Modify the `/etc/newunix` script so that it will accept the name of your driver as an argument. An example of such an argument is `/etc/newunix` *mydevice*. Also modify the script so that the user can specify `no`*mydevice* to uninstall your driver.

Next you should run `/etc/newunix` *mydevice*. After your *mydevice* install script finishes execution, your driver object file, master script file, and optional scripts should be in the appropriate directories for `autoconfig`.

## Running autoconfig

After you run `/etc/newunix`, the files that `autoconfig` needs to link your driver into the kernel should be in place. You can now run `autoconfig`. Do not use the `-a` option. By not specifying the `-a` option, you tell `autoconfig` to build a new kernel.

If you provided the necessary files and information, `autoconfig` will link your driver or module into the new kernel. If you run `autoconfig` with the `-I` option, `autoconfig` executes all init scripts in the `/etc/init.d` directory. If you run `autoconfig` with the `-S /etc/startup` option, `autoconfig` adds your startup script to the list of startup scripts in `/etc/startup`.

You can now powerdown your system and install your hardware. When you turn the system back on, your new kernel should boot and you can begin to perform I/O to your device.

If your hardware is already installed, then shutdown the system and reboot. Your new kernel should boot and you can begin to perform I/O to your device.

## Customer installation of your driver

Once you have successfully debugged and tested your driver, you are ready to distribute your driver to your customers. To install your driver, your customers should use the `finstall` utility.

For information on how to install your driver from your distribution floppy to a customer's A/UX system disk, see Chapter 14. Chapter 14 describes how your customers can use `finstall` to install your driver object file into `/etc/install.d/boot.d`, your install script into `/etc/install.d`. After your customers install your driver object file and install script onto their system, they can execute `/etc/newunix`, run `autoconfig`, and then powerdown the system and install the hardware. When they turn the system back on, the new kernel should boot, and your customers can perform I/O to your device.

# Chapter 13

# Using Autoconfiguration

For end users who buy and install a new device in the Macintosh II, the autoconfiguration process is used to link new drivers into the kernel. This process includes using `finstall, /etc/newunix, autoconfig`, and then rebooting the system. If you're developing device drivers, however, you may want to automate the normal autoconfiguration sequence by using a makefile to install and test your driver.

This chapter describes how you can use autoconfiguration in this way for developing drivers. This chapter uses a specific example to illustrate how to create the various script files used with your driver, and to illustrate how to write a makefile that allows you to recompile and add your driver to the kernel with one command. Before you start, make sure that you have read Chapter 12 and understand how autoconfiguration works, and that you know what script files you must write.

## The sample TEST driver

The sample driver used as an example in this chapter is named TEST. The TEST driver is a character device driver that controls a slot card. The source code for the TEST driver is shown in this section. The following sections show the master script file, install script, uninstall script, startup script, modified `/etc/newunix` file, makefile, and loadfile that can be used with the `TEST` driver.

```
#include <sys/sysmacros.h>
#include <sys/reg.h>

extern int TESTcnt;
extern int TESTaddr[];


TESTopen (dev,flag)

dev_t dev;

{

int maj, min;

      maj = major(dev);

      min = minor(dev);

      printf(" in TESTopen now \n");

      printf(" The major number is %d \n", maj);

      printf(" The minor number is %d \n", min);

      return(0);

}
```

```
TESTclose (dev)
dev_t dev;
{
      printf(" in TESTclose now \n");
}
TESTinit()
{
      int i;

      printf (" in TESTinit \n");
      /* Recall that if you specify the "a" flag in the Device Information
         line of the master script file, autoconfig creates the variables
         prefixcnt and prefixaddr.                                        */
      /* In this specific example, TESTcnt contains the number of slot
         cards with board id 99 in the system.  TESTaddr[] contains the
         slot number of each slot card in the system that the TEST driver
         controls.                                                        */
      for (i = 0; i<TESTcnt; i++)
      printf (" TESTaddr [%d] is in slot %d\n", i, TESTaddr[i]);
}
TESTint (args)
struct args *args;
{
printf(" Slot card generating interrupt is in slot %d\n", args->a_dev);
}
```

## The TEST master script file

You use the master script file to link the TEST driver into the kernel. The install script /etc/install.d/TEST is used to create the master script file and to place the master script file in /etc/master.d. The master script file for the TEST driver is as follows:

```
id 99 1
if . include slots
acvs        1     TEST        -     2     1
```

The first line of the `TEST` master script file specifies the board id of the slot card (99), and the version number (1). When `autoconfig` is run, `autoconfig` looks for slot cards with board id 99 that might exist in the system. If any slot cards with board id 99 exist in the system, `autoconfig` will include the `TEST` module in the kernel.

The second line instructs autoconfig to include the slots module in the kernel if the `TEST` module is included. Remember that for device drivers other than slot device drivers and device drivers that are not included by any other master script file, this line must contain "`include .`" to include the module into the kernel.

The third line instructs `autoconfig` to create the `TESTcnt` and `TESTaddr` data structures (a), specifies that the `TEST` module is a character device driver(c), and specifies that `TEST` receives interrupts from a slot device (`vs`). This line also instructs `autoconfig` to set up the appropriate entry of the slot interrupt vector table to point to the `TESTint` routine.

The third line also specifies that `TEST` receives one interrupt per controller, as do all slot device drivers. The driver prefix is `TEST`, so `autoconfig` will look for routines with this prefix to create entry points in the cdevsw structure for this module.

The software major number will be assigned by `autoconfig` (-). The `TEST` module supports up to two devices per slot card, and interrupts at priority level 1.

## The TEST startup script

You can provide a startup script with your device to perform various functions at boot time. The install script `/etc/install.d/TEST` is used to create the startup script file and to place the master script file in `/etc/startup.d`.

You must run `autoconfig` with the `-S /etc/startup` option if you want `autoconfig` to add your startup script to the list of startup programs in `/etc/startup`. The startup script file for the `TEST` driver is as follows:

```
#       startup script for TEST

PATH=/bin:/usr/bin:/etc:/usr/etc
scriptname=$0
         name=TEST

#        Your startup script is passed a number of flags
#        Refer to Chapter 12 for a description of these flags

#        Initialize the minor number, so that each device
```

```
#       has a unique minor number
minor=0

while test -n $1

do

      case $1 in

#                   For -M flag: Save the major number
      -M)   shift

            major=$1
            echo "The major number is $major"
            ;;

#                   For -C flag: echo the number of cards for
#                                this driver in the system
      -C)   shift

            echo "$1 card(s) installed for TEST driver"
            ;;

#                   For -D flag: echo the number of devices per
#                                card
      -D)   shift

            echo "$1 device(s) per card"
            ;;

#                   For -S flag: Create the device file

#                                Each device file is named
#                                /dev/TESTslotnumber and
#                                is given a unique minor
#                                number.
      -S)   shift

            mknod /dev/TEST$1 c $major $minor
            minor=`expr $minor + 1`
            echo "There is a card in slot $1 for TEST"
            ;;


#                   Print error for all other flags

      *)    echo "$scriptname: Unexpected argument $1"
            exit 1
            ;;

esac

shift
```

```
done
#       End of TEST startup script
```

## The TEST install script

The install script for the TEST device driver is located in /etc/install.d. You use the /etc/newunix command to execute the TEST install script. The TEST install script copies the TEST object file from /etc/install.d/boot.d/TEST to /etc/boot.d. The TEST install script also creates the TEST master script file, and creates a startup script.

The TEST install script is as follows:

```
#       /etc/install.d/TEST

PATH=/bin:/usr/bin:/etc:/usr/etc

        name=TEST

#
#       Install the driver object file
#
        cp /etc/install.d/boot.d/$name /etc/boot.d
        chmod 644 /etc/boot.d/$name

#
#       Install the driver master script file
#
        echo 'id 99 1' > /etc/master.d/$name
        echo 'if . include slots' >> /etc/master.d/$name
        echo 'acvs   1   TEST   -   2   1' >> /etc/master.d/$name

        chmod 644 /etc/master.d/$name

#       end of TEST install script
```

## The TEST uninstall Script

The uninstall script for the TEST device driver is located in /etc/uninstall.d. You use the /etc/newunix command to execute the TEST uninstall script. The TEST install script removes the files related to TEST in the directories used by autoconfig.

The TEST uninstall script is as follows:

```
#       /etc/uninstall.d/TEST

PATH=/bin:/usr/bin:/etc:/usr/etc

        name=TEST

#
#       Delete the driver object file
#
        rm -f /etc/boot.d/$name

#
#       Delete the driver master script file
#
        rm -f /etc/master.d/$name

#
#       Delete the driver startup file
#
        rm -f /etc/startup.d/$name

#       end of TEST uninstall script
```

## Modifying /etc/newunix

Remember that you need to either supply your customers with a modified version of `/etc/newunix`, or give them directions on how to modify the file themselves.

The arguments currently available that a user can specify to `/etc/newunix` are: basic networking (`bnet`), Network File System (`nfs`), A/UX toolbox (`toolbox`), non-networking (`nonet`), no toolbox capabilities (`notoolbox`).

For each module specified in the command line, `/etc/newunix` invokes a script corresponding to that module. The scripts for individual modules are located in `/etc/install.d`.

The scripts in `/etc/install.d` set up the files that `autoconfig` needs in order to configure that module into the kernel. After the script executes, the appropriate files have been placed in `/etc/master.d`, `/etc/boot.d`, `/etc/startup.d`, and `/etc/init.d` that `autoconfig` will use to link the module into the kernel.

You must modify `/etc/newunix` to include processing of your install and uninstall scripts. Add a line to the `case` statement that will accept the name of your driver as a parameter. When this parameter is specified, you should execute the install script for your driver.

You should also add a line that accepts the prefix no and the name of your driver (for example, noTEST). When this parameter is specified, you should execute the uninstall script for your driver.

The following is a modified version of /etc/newunix that works with the install and uninstall scripts of TEST.

```
#
#       Modified version of /etc/newunix that also accepts TEST as an
#       argument

PATH=/bin:/usr/bin:/etc:/usr/etc
case $1 in
        nonet)  /etc/uninstall.d/BNET
                /etc/uninstall.d/ae6
                /etc/uninstall.d/nfs
                ;;

        bnet)   /etc/uninstall.d/nfs
                /etc/install.d/BNET
                /etc/install.d/ae6
                ;;

        nfs)    /etc/install.d/BNET
                /etc/install.d/ae6
                /etc/install.d/nfs
                ;;

        toolbox)
                /etc/install.d/toolbox
                ;;

        notoolbox)
                /etc/uninstall.d/toolbox
                ;;
#
#       Add a line that checks for your device name here
#
       TEST)
             /etc/install.d/TEST
                ;;
       NOTEST)
             /etc/uninstall.d/TEST
                ;;

        *)      echo "Usage:  $0 <system>"
                echo "  where <system> is one of nonet, bnet, nfs, toolbox
or no toolbox, TEST or noTEST"
                exit 1
```

```
                ;;
esac
#       end of modified /etc/newunix
```

## Using makefiles

After you write your driver, master script file, and optional script files, you can create and run a **makefile**. This file contains user-specified commands that are processed according to built-in rules found in the `make` utility. For more information about this utility, see "Using make" in *A/UX Programming Languages and Tools Volume 2*, and `make`(1).

Your makefile should contain four commands:

- A command that compiles your driver.
- A command that copies your driver into the `/etc/install.d/boot.d` directory.
- A command that executes the modified `/etc/newunix` command.
- A command that executes the `autoconfig`(1M) utility.

In addition, you can create a **loadfile** to hold slot ROM information. From the `autoconfig` command in your makefile, you can specify whether or not this file will be read instead of the slot ROMs for your card.

## Creating a loadfile

During development of your driver, you should run `autoconfig` from the system directly to create a new kernel.During development of NuBus cards, you can test your slot device driver independently of your hardware. If your slot card is not ready for testing with your software driver, or if you have not yet installed the slot ROMs on your card, you can use a loadfile to begin testing your driver. In the place of slot ROMs, you must create a loadfile to hold slot ROM information.

The loadfile is an ASCII file which contains the following information:

*slot-number board-ID version-number*

For example, this loadfile

```
11 99 1
```

specifies slot 11, board ID 99, and version number 1. To use a loadfile, specify the `-L` *loadfile* option to `autoconfig`. When you specify this option,`autoconfig` reads the specified file for device information instead of reading the slot ROMs.

If you create a kernel using the `-L` *loadfile* option to `autoconfig`, then you must use the `-n` option on the `launch` command line to boot this new kernel. The `launch -n` option forces launch to set the AUTO_OK flag, regardless of whether the slot card for your driver is present or not. Note that you should only use a loadfile and the `launch -n` option during driver development and testing of your software driver. When you are ready to test your software driver with your hardware, then you do not need to use a loadfile or the `launch -n` option.

## The sample TEST makefile

You can create a makefile to automate the process of compiling and linking your driver to the kernel. The sample makefile used with the `TEST` driver is shown here:

```
testunix:          /newunix /etc/install.d/boot.d/TEST loadfile
                   /etc/newunix TEST
                   autoconfig -L loadfile -I -S /etc/startup -o /testunix
/etc/install.d/boot.d/TEST:    TEST.o
                               cp TEST.o /etc/install.d/boot.d/TEST
TEST.o:            /bin/cc -c TEST.c
```

The rule `testunix:` checks for the `/newunix file`, the driver file TEST, and the loadfile, and if present, runs `/etc/newunix` and then `autoconfig`. If the driver file `TEST` is not present in the `/etc/install.d/boot.d` directory, `make` executes the `/etc/install.d/boot.d/TEST` rule.

The `/etc/install.d/boot.d/TEST` rule depends on `TEST.o`. If `TEST.o` does not exist, the `TEST.o` rule is executed. If `TEST.o` does exist, `make` executes the command on the following line. This command copies `TEST.o` into the `/etc/install.d/boot.d` directory.

The rule defined by `TEST.o` compiles the `TEST.c` driver code.

After the `/etc/install.d/boot.d/TEST` file has been updated, `make` executes the next statement on the line following the `testunix` statement. This line is a command to run `/etc/newunix`.Remember this must be a modified version of `/etc/newunix` that has been modified to accept the name of your driver as an argument.

The TEST argument to `/etc/newunix` causes the `/etc/install.d/TEST` script to be executed. The TEST install script copies the driver object file `/etc/install.d/boot.d/TEST` to `/etc/boot.d`. The TEST install script also creates the master script file `/etc/master.d/TEST`.

The make utility then executes autoconfig. The options to autoconfig are explained in the following paragraphs.

The -L flag means that autoconfig reads the loadfile instead of searching the slots for a device. If you do not specify a full pathname, autoconfig looks in the current working directory for the file named loadfile.

The -I flag instructs autoconfig to call the init scripts in /etc/init.d for all modules included in the new kernel.

The -S /etc/startup option instructs autoconfig to create a list of startup scripts for modules in the new kernel. If you have supplied a startup script for your driver, the name of your startup script is put in the specified file (/etc/startup). When the system is rebooted, your startup script will be executed.

The -o flag changes the default output file from /unix to /testunix.

## Creating a new kernel that includes your driver

Using the sample makefile in the previous section, you can create a new kernel that includes the TEST driver by typing

```
make testunix
```

The make program executes the TEST makefile. Make outputs several on-screen messages, including error messages if make finds any errors in the makefile. Warning messages preceded by the string Expect a warning message can be ignored.

If the TEST driver is successfully added to the kernel, autoconfig prints a table of existing modules, which should now include the TEST driver.

The file /testunix now contains the TEST driver. You should now back up /unix (for example, by using cp /unix /oldunix), and then move the new kernel to /unix (for example, by using mv /testunix /unix). If your hardware is already installed, shut down the system and reboot.

If your hardware is not yet installed, then you should power down the system and install the hardware at this time.

After rebooting the system, the TEST driver is available to perform I/O.

## Performing I/O with the TEST driver

After using the make process and rebooting the new kernel, you are ready to debug the TEST driver. A sample program that opens the TEST driver is shown below:

```
#include <fcntl.h>
#include <errno.h>

main()
{

int fd;

printf(" Begin testing driver \n");

if ((fd = open("/dev/TEST11", O_RDWR)) == -1)
{
perror(" Error in open, errno message ");
exit(1);
}

close(fd);

}
```

You also need to make sure the device file for your driver file has been created. Typically your driver init or startup script creates this file. The `TEST` startup script created the `/dev/TEST11` device file.

After compiling and executing this sample program, the following output is produced on the terminal:

```
Begin testing driver
In TESTopen now
The major number is 9
The minor number is 0
```

This chapter showed how to create a simple character device driver for a slot card, by using the `TEST` driver as an example. The master script file, startup script, install script, uninstall script, modified `/etc/newunix` file, makefile, and loadfile for the `TEST` driver were also shown.

You should now know how to begin writing the device driver for your device, and how to create the files used during autoconfiguration to add your driver to the kernel.

# Chapter 14

# Preparing Your Driver For Customer Distribution

After writing and successfully testing your driver, you need a procedure your customers can use to install your driver onto their A/UX system. In this chapter, you'll learn how to prepare your distribution floppy disk with the files needed to install the software for your device.

Apple Computer has designed a standardized installation procedure to install third-party software called `finstall`. `finstall` is a Bourne shell script that you can use to install software from one or more floppy disks. The floppy disks should contain an A/UX mountable file system with various files on it, including a cpio archive containing files to be installed, and optional `preinstall` and `postinstall` shell scripts or executable programs.

`finstall` is intended to provide a simple, common, and consistent user interface for installing software on an A/UX system.All third-party vendors should use `finstall`. Apple also uses `finstall` for A/UX software installation.

A customer can install your software by simply typing the `finstall` command at the shell prompt. `finstall` prompts the user for certain information, such as which drive (right hand or left hand) the floppy disk is inserted in. Each question has a standard default, so you can make the installation process completely automatic.

Your customer runs `finstall` to install your driver object file, install script, and uninstall script into the appropriate directories of their A/UX system.

After your customer runs `finstall`, your driver object file should be located in the `/etc/install.d/boot.d` directory of the A/UX system disk. In addition, your install script should be located in the `/etc/install.d` directory, and your uninstall script in the `/etc/uninstall.d` directory.

Your customer then needs to run `/etc/newunix`. Remember that you need to either supply a modified version of `/etc/newunix` for your customers, or give them directions on how to modify the file themselves.

When your customer runs your modified version of `/etc/newunix`,the customer must specify the name of your device as an argument. `/etc/newunix` will then execute your install script.

The install script for your device should copy the driver object file in /etc/install.d/boot.d into /etc/boot.d. Your install script should also create the master script file and other optional script files of your driver, and place these scripts in the appropriate directories that `autoconfig` needs to link your driver into the kernel.

After your customer runs your modified version of `/etc/newunix`, the appropriate files should now be in `/etc/master.d`, `/etc/boot.d`, `/etc/startup.d`, and `/etc/init.d`, which are the directories `autoconfig` uses when linking the module into the kernel. After making sure the current system has been backed up, then the customer should run `autoconfig`.

`Autoconfig` links your module into the new kernel, and puts the new kernel in `/unix`. Your customer should now powerdown the system and install the hardware. After installing the hardware, your customer can turn the power to the system on. When the A/UX system begins initialization, the new kernel that includes your driver will be booted. Your customer can now perform I/O to your device.

## Giving out finstall to your customers

`finstall` is a Bourne shell script that should be located in `/usr/bin`. Not all customers have `finstall` on their systems however, as `finstall` was not distributed with A/UX Release 1.0.

Therefore, Apple is supplying a copy of `finstall` with the Device Drivers Kit. You should include a copy of `finstall` on the distribution disk for your software.

If `finstall` is not installed on the user's system, the message

```
finstall: Command not found.
```

will be displayed on the user's monitor. In this case the user needs to copy `finstall` from your software distribution disk to the `/usr/bin` directory of their system. The user can use the following command to copy `finstall` to their system:

```
cpio -icuvm /usr/bin/finstall < /dev/rfloppy0
```

The user should get the message

```
/usr/bin/finstall
```

if `finstall` was successfully copied to the `/usr/bin` directory.

This listing shows the way a customer would install your software, using the A/UX Device Drivers Kit as an example.

```
% finstall

finstall: This is the finstall program. It installs software from a
finstall: floppy disk onto your system. It will give you a chance to
finstall: see what software is being installed and how much disk
finstall: space it will need before anything is installed.

finstall: You may at anytime stop the finstall procedure by giving
finstall: it an interrupt, which is normally the CTL-C key.

finstall: Press RETURN when ready to proceed: (RETURN)

finstall: Use the left or right floppy disk drive? [default: right] (RETURN)
```

```
finstall: Insert the installation floppy number 1 into the right hand drive.
finstall: Press RETURN when ready to proceed: (insert floppy and press RETURN)

finstall: Now mounting the installation floppy....

finstall: Now checking that the required files are on the floppy...

finstall: The software on this floppy is <Device Drivers>
finstall: It is from <Apple Computer>
finstall: And it is version <v1.0>
finstall: The vendor has supplied the following description:
```

(A short two paragraph description of the A/UX Device Drivers Kit is shown)

```
finstall: Under what directory should the software be installed? [default:
/usr/src/device_drivers]  (RETURN)

finstall: Now calculating the disk space needed to install the software.

finstall: This may take a few minutes....


finstall: You have 20160 blocks on the installation point <cs>.
finstall: You will use 5200 blocks to install the software, leaving 14960
blocks free.

finstall: Do you want to see what files will be installed before they are
actually installed? [default: yes]  (RETURN)
```

(finstall displays a list of all the files that will be installed)

```
finstall: This is your last chance to stop before actually installing the software.
finstall: Do you want to proceed with the installation? [default: yes]
(RETURN)
```

(finstall proceeds to install the files)

## An overview of finstall

Your distribution floppy disk can contain a number of files as shown in Figure 14-1.

**Figure 14-1.**
**Files on the distribution floppy disk**

When the user executes `finstall`, `finstall` performs a number of functions. `finstall` first checks whether the `/etc/finstallrc` or `.finstallrc` file exists on the customer's A/UX system. A user can control the default options for `finstall` by using these two files.

`finstall` also checks that the user is the superuser (root), then mounts the floppy disk and creates the following directories if they do not exist:

```
/etc/finstall.d
/etc/finstall.d/vendorname
/etc/finstall.d/vendorname/softwarename
/etc/finstall.d/vendorname/softwarename/versionname
```

The *vendorname*, *softwarename*, and *versionname* are taken from the names specified in their respective files on the distribution floppy disk.

`finstall` uses the `sequenceno` and `sequenceof` files to determine if this floppy disk is part of a multi-floppy disk set. If so, `finstall` verifies that the floppy disk has the correct sequence number specified in the `sequenceno` file.

The `installpoint` file contains the default absolute pathname of where to install the files in `cpiodata` or `cpiodata.z`. `finstall` prompts the user for a different installation point, allowing the user to override the default specified in `installpoint` if desired. If you do not provide an `installpoint` file,`finstall` uses the current working directory as a default installation point.

`finstall` calculates the amount of disk space needed to install the files, and if not enough space is available, lets the user decide whether to quit or continue. Just before copying the files from the `cpiodata` or `cpiodata.z` archive, `finstall` executes the `preinstall` script (if it exists) on the floppy disk.

`finstall` also creates a list of the files installed from the cpio archive. `finstall` puts this list of installed files in the file `/etc/finstall.d/`*vendorname*`/`*softwarename*`/`*versionname*`/installedfiles`.

`finstall` also copies either the absolute pathname specified by the `installpoint` file on the floppy disk or the installation point specified by the user to the `/etc/finstall.d/`*vendorname*`/`*softwarename*`/`*versionname*`/installpoint` file.

`finstall` then installs the files from the `cpiodata` or `cpiodata.z` archive. The files copied from the floppy disk are placed in either the current working directory or the directories indicated by the information from the `installpoint` and `installedfiles` files.

If the software distribution is on more than one floppy disk, `finstall` continues with the installation procedure for the next floppy disk. After all files have been installed, `finstall` executes the `postinstall` script (if it exists) on the last floppy disk of the software distribution. If your `postinstall` script creates any files, then your `postinstall` script should update the `/etc/finstall.d/`*vendorname*`/`*softwarename*`/`*versionname*`/installedfiles` file accordingly.

You can set various default options for `finstall`. For example, you can specify that `finstall` prompt for an installation point, or specify that `finstall` use the current working directory as the installation point and not prompt for an installation point. These default options are listed in the following section.

## Setting defaults for finstall on your A/UX system

You can use the `/etc/finstallrc` and `.finstallrc` to specify default options used with `finstall`, such as whether `finstall` should prompt the user for certain information, or whether `finstall` should use a default value.

You can set these options by either placing them in the `/etc/finstallrc` file or in `.finstallrc` in the current working directory. The options that you can specify are Bourne shell "set" type options. The default values set by `finstall` are as follows:

```
CON_TRIES=5
CTL_ALLOWRC=1
CTL_ASKDRIVE=1
CTL_ASKINSTALL=1
CTL_CHECKSPACE=1
CTL_TAKEDEFAULT=0
```

The settings of these options are explained in the following paragraphs.

`CON_TRIES` This option specifies the number of tries a user is allowed during an attempt to give an acceptable answer to a prompt. If a user uses all of the tries, `finstall` quits. This number should be a positive integer value.

`CTL_ALLOWRC` This option specifies whether `finstall` should use the `.finstallrc` file in the working directory. `CTL_ALLOWRC` can be set as follows:

| | |
|---|---|
| `CTL_ALLOWRC == 0` | Do not use a .finstallrc file in the current working directory. |
| `CTL_ALLOWRC != 0` | Do use a .finstallrc file in the current working directory if it exists. |

`CTL_ASKDRIVE` This option specifies whether `finstall` should prompt for the drive that will be used to install the software from. `CTL_ASKDRIVE` can be set as follows:

| | |
|---|---|
| `CTL_ASKDRIVE == 0` | Don't ask which drive is to be used; assume that it is the right drive. |
| `CTL_ASKDRIVE != 0` | Ask if the right or left drive is to be used. |

`CTL_ASKINSTALL` This option specifies whether `finstall` should prompt for the installation point on the user's system where the software will be installed. `CTL_ASKINSTALL` can be set as follows:

| | |
|---|---|
| `CTL_ASKINSTALL == 0` | Don't ask for an installation point; assume current working directory. |

CTL_ASKINSTALL != 0 Ask for an installation point.

CTL_CHECKSPACE — This option specifies whether `finstall` should check if there is enough space on the installation point to install the software. CTL_CHECKSPACE can be set as follows:

CTL_CHECKSPACE == 0 Do not check for space on the install point.
CTL_CHECKSPACE != 0 Check that there is enough space on the install point to install.

CTL_TAKEDEFAULT — This option specifies whether `finstall` should use default answers. CTL_TAKEDEFAULT can be set as follows:

CTL_TAKEDEFAULT == 0 Whenever an answer is prompted for, read it from the controlling TTY device.

CTL_TAKEDEFAULT != 0 Print the question on the screen, but do not wait for an answer. In this case, the default answer is used.

## Files that are located on the finstall floppy disk

This section describes the files involved in the `finstall` installation procedure.

`cpiodata` — The cpio archive that contains the files to be installed. You must create this archive with the `-c` option of `cpio`. For example, if you have two directories containing various files that you wish to create, you can type the following command line:

```
% find dir1 dir2 -depth -print | sort | \
  cpio -oc > cpiodata
```

Note that you can use either `cpiodata` or `cpiodata.z`, but not both.

`cpiodata.z` — This file is a packed version of `cpiodata`. To create the packed version, run the pack utility on the `cpiodata` file. An example of this is shown below:

```
% find dir1 dir2 -depth -print | sort | \
  cpio -oc > cpiodata ; pack cpiodata
```

`description` This file contains simple ASCII text. The text consists of a paragraph or two describing the software. Because this text will be displayed for the user, you should make the text as descriptive as possible. For example, the file may contain something like the following:

```
This is the MARSH Corporation "Ally Gator" video board
driver installation software. When installed, this
software will build a new A/UX kernel to support the
Ally Gator video board.
```

`diskspaceneeds` This file contains one line of ASCII text. This text contains three numbers that assist the `finstall` script in calculating how much disk space is needed for installing the software on this floppy disk. Normally, if this file is not present, `finstall` calculates the disk space needed based on the size of the `cpiodata` file. If this file is present, `finstall` takes the size of the `cpiodata` file and uses the numbers from this file to adjust the `diskblocksneeded` value.

For multi-volume releases, each floppy disk can contain a `diskspaceneeds` file that represents the remaining data to install. For example, the `diskspaceneeds` file for the first floppy disk contains the size of the entire package, and each following floppy disk contains a smaller value.

Of the three numbers in the `diskspaceneeds` file, the first number is the number of disk blocks to add or subtract from the size calculated from the `cpiodata` file. A positive or unsigned number is added to the size of the `cpiodata` file, while a negative number is subtracted. Leave this number as 0 if you do not wish to use it.

The second number is a percentage to increase or decrease the number of blocks needed, based again on the size of the `cpiodata` file. For example, if the second number is 30, `finstall` increases the total number of blocks needed by 30%. If the number is -30, then `finstall` decreases the total number of blocks by 30%. Thus, if the size of the `cpiodata` file is 100 blocks, and if the second number is 30, then the total size required is 130 blocks. Leave this number as 0 if you do not wish to use it.

You can use both the first and second numbers together.

The third number, if not zero, overrides the first and second numbers. The third number is the absolute number of blocks to use as the disk-space-required value. The size of the `cpiodata` file is ignored in this case. Leave this number as 0 if you do not wish to use it.

`preinstall` This file is an executable program or shell script that is executed right before the cpio utility installs the files. You might use the `preinstall` script to save files or directories that may be overwritten by the installation process. The `preinstall` script has five arguments passed to it. These arguments are exactly the same as those passed to `postinstall`, and are as follows:

- arg1 The root mount point of the floppy disk.
- arg2 The installation point of the software.
- arg3 The full pathname of the version directory of the software. The version directory is a directory located under `/etc/finstall.d`. It has this format:

  `/etc/finstall.d/`*`vendorname/softwartename/versionname`*.

  Files pertaining to the installation of this version of software are kept in the version directory.
- arg4 The full pathname of the filename of the `installedfiles` file under the *versionname* directory. The `installedfiles` file contains a list of files, one per line, that are installed. The files in the `cpio` archive are automatically placed in this file. If either the `preinstall` or `postinstall` program or shell scripts install or create files, then those filenames should be placed in this file also. The basename of this file is always `installedfiles`.
- arg5 The full pathname of the filename of the `installpoint` file under the *version* directory. Files in the `cpio` archive can be either absolute pathnames or relative pathnames. The `finstall` script asks the user for a pathname to install under (the default is the current working directory). This pathname is known as the install point. This file contains the install point.

This program or shell script should return 0. A nonzero return value causes `finstall` to prompt the user for permission to continue.

`postinstall` This file is an executable program or shell script that is executed right after the `cpio` utility has installed the files.You might use the `postinstall` script to create links or for other post-installation functions. The `postinstall` script has five arguments passed to it. These arguments are exactly the same as those passed to `preinstall`. This program or shell script should return 0. A nonzero return value causes `finstall` to prompt the user for permission to continue.

`sequenceno` This file contains one line of ASCII text. A number indicating the sequence number of the floppy disk in a multi-floppy disk set is the sole contents of the line. For single floppy disk installations, this file can be omitted. If this file is present, the `sequenceof` file must also be specified.

`sequenceof` This file contains one line of ASCII text. A number indicating the number of the floppies in a multi-floppy disk set is the sole contents of the line. For single floppy disk installations, this file can be omitted. If this file is present, the `sequenceno` file must also be specified.

`vendorname` This file contains the name of the vendor of the software being installed. This name should be a System V UNIX directory name and must adhere to the naming conventions (14 characters or less, no embedded slashes). Embedded blanks are allowed – the `finstall` script carefully quotes all use of the *vendorname* directory name. The *vendorname* directory is under the `/etc/finstall.d` directory, as shown below.

`/etc/finstall.d/`*vendorname*

You should carefully chose your *vendorname* name, and keep it consistent for all your software products. For example, software products from Apple Computer use the vendor name Apple Computer for all products.

`softwarename` This file contains the name of the software being installed. This name should be a System V UNIX directory name and must adhere to the naming conventions (14 characters or less, no embedded slashes). Embedded blanks are allowed – the `finstall` script carefully quotes all use of the *softwarename* directory name. The *softwarename* directory is under the *vendorname* directory, as shown below.

`/etc/finstall.d/`*vendorname/softwarename*

`versionname` This is the version of the software being installed. This name should be a System V UNIX directory name and must adhere to the naming conventions (14 characters or less, no embedded slashes). Embedded blanks are allowed - the `finstall` script carefully quotes all use of the *versionname* directory name. The *versionname* directory is under the *vendorname* and *softwarename* directories, as shown below.

`/etc/finstall.d/`*vendorname/softwarename/versionname*

`installpoint` (floppy disk file) This file contains the default absolute pathname of the home directory from where the files will be installed.

`installpoint` (system disk) This file contains the actual installation point used. This is either identical to the pathname specified by the `installpoint` file on the distribution floppy disk, or the pathname specified by the user.

`installedfiles` (system disk)

This file contains the list of files that were installed.This file is not supplied on the distribution floppy disk. The `installedfiles` file is created by `finstall` in the file `/etc/finstall.d/`*vendorname/softwarename/versionname*`/installedfiles`.

The file names can be either absolute or relative path names. If relative pathnames are used, then you must provide the `installpoint` file also.

If your `postinstall` script installs or creates any files, your `postinstall` script should modify this file accordingly.

# Appendix A

# Driver Interface Routines

This appendix is a reference section for the driver routines that are invoked through the bdevsw and cdevsw tables. These routines provide the driver interface to the kernel. Some of these routines are found in both block and character device drivers; some are specific to only block device drivers or only character device drivers. Note that this appendix does not include descriptions of the rouitnes used in streams device drivers. Refer to Chapter 6,"Streams Device Drivers", for a description of the routines used by streams device drivers.

The following routines are found in this appendix:

- *driver*`open`—prepares the device for I/O. Both block and character device drivers supply a *driver*`open` routine.
- *driver*`close`—performs device close operations. Both block and character device drivers supply a *driver*`close` routine.
- *driver*`read`—reads data from a device. Only character device drivers supply a *driver*`read` routine.
- *driver*`write`—writes data to a device. Only character device drivers supply a *driver*`write` routine.
- *driver*`ioctl`—performs control operations or other device-dependent operations on a device. Only character device drivers supply a *driver*`ioctl` routine.
- *driver*`strategy`—schedules the transfer of data between the kernel buffer cache and a device. Only block device drivers supply a *driver*`strategy` routine that is directly invoked by the kernel. However, a character device driver can indirectly call a *driver*`strategy` routine by using the kernel routine `physio()`.
- *driver*`print`—prints error messages to the user on the system console. Only block device drivers supply a *driver*`print` routine.

Drivers can also provide an initialization routine called *driver*`init`. The kernel calls *driver*`init` routines during system initialization. Refer to the section "Including Device Information" in Chapter 12 for more information on the *driver*`init` routine.

Slot device drivers must provide an interrupt routine called *driver*`int`. Chapter 9 describes the *driver*`int` routine for slot device drivers. Most other drivers also supply an interrupt routine. For information on how other drivers handle interrupts, refer to the section "Interrupt Handling by Your Driver" in Chapter 1.

## Return values of driver interface routines

A/UX device driver `open`, `read`, `write`, and `ioctl` routines must return either 0 for success or an error number for failure. Error numbers referred to as *errno* are defined in the header file <`sys/errno.h`> and are listed in Appendix B.

## A summary of the driver interface routines

The rest of this appendix describes the driver interface routines. Entries are listed in alphabetic order and contain the following:

- the name of the routine
- a synopsis of the routine
- the arguments to the routine
- a description of what the routine does
- the values returned from the routine
- where to look for more information

## Name

close—perform device close operations

## Synopsis

`void` ***driver*`close(`*dev, flag*`)`

`dev_t` *dev*;

`int` *flag*;

where

- *dev* is the device number.
- *flag* is a flag from the *oflag* field of the open system call (see open(2) in the *A/UX Programmer's Reference*). These flags correspond to the flag values in a file descriptor data structure (the `f_flag` field in the header file `<sys/file.h>`).
- *driver* is the device prefix.

## Description

The ***driver*`close`** routine is used to remove the connection between the physical device and the driver. Typical functions of a ***driver*`close`** routine include reinitializing driver data structures and device hardware. The kernel calls `close` only on the last close of the device; that is, if no other processes have the device open. The ***driver*`close`** routine should take the appropriate actions to make the device available to be opened later.

## Return values

None.

### See also

For block devices, see "The driverclose routine" in Chapter 3.

For character devices, see "The driverclose routine" in Chapter 4.

For terminal devices, see "The close routine" in Chapter 5.

For Streams devices, see "The close routine" in Chapter 6.

For Streams terminal devices, see "The close routine" in Chapter 7.

## Name

ioctl—perform control operations and other device-dependent operations

## Synopsis

`int` ***driver*`ioctl` (*dev, cmd, addr, mode*)

`dev_t` ***dev***;

`int` ***cmd***, ***mode***;

`caddr_t` ***addr***;

where

- ***dev*** is the device number.
- ***cmd*** is a command argument indicating the type of operation to be done. The value of ***cmd*** corresponds to the ***req*** parameter specified by the user in the `ioctl`(2) system call. The value of ***cmd*** is driver dependent (see Section 7 of the *A/UX System Administrator's Reference* for ioctl command values of different drivers).
- ***addr*** is a pointer to a buffer containing data copied in from the ***arg*** parameter specified by the user, or is a storage area to place data to be copied out to the user in the ***arg*** parameter.
- ***mode*** is an argument that contains values set when the device was opened. The driver can use ***mode*** to check whether the device was opened for read or write.
- ***driver*** is the device prefix.

## Description

You can use a ***driver*`ioctl`** routine to perform device-specific or driver-specific commands. For example, you could use your ***driver*`ioctl`** routine to perform control operations on your device, to get status from your device, or to change the configuration of your device. Common uses of the ***driver*`ioctl`** routine are to perform miscellaneous activities such as rewinding a tape or initializing a disk. Only character device drivers provide a ***driver*`ioctl`** routine.

## Return values

If your *driver*`ioctl` routine successfully performs the request, your *driver*`ioctl` routine should return zero to the kernel. If your *driver*`ioctl` routine is unable to successfully perform the request, your *driver*`ioctl` routine should return an `errno` value to the kernel, indicating the reason the request failed.

### See also

For character devices, see "Performing Control and Miscellaneous Functions on a Device" in Chapter 4.

For terminal devices, see "The ioctl routine" in Chapter 5 and `termio`(7) in the *A/UX System Administrator's Reference*.

For streams terminal devices, see "The ioctl routine" in Chapter 7.

`copyin`(kernel) in Appendix B.

`copyout`(kernel) in Appendix B.

## Name

open—prepare the device for I/O

## Synopsis

`int` ***driver*`open(`*dev, flag, ndevp*`)`

`dev_t` *dev*, **ndevp*;

`int` *flag*;

where

- *dev* is the device number.
- *flag* corresponds to the flag values in a file descriptor data structure (the *f_flag* field in the header file `<sys/file.h>`).
- *ndevp* is a pointer to a `dev_t`, which is used in clone open operations for character devices. Only character device drivers are passed the *ndevp* parameter.
- *driver* is the device prefix.

## Description

The ***driver*`open` routine is used to prepare the device for I/O. Typical functions of a ***driver*`open` routine include validating the device number, and performing device-dependent open operations. The ***driver*`open` routine should open the file according to the *flag* parameter and prepare the device for data transfer.

## Return values

If your ***driver*`open` routine successfully opens the device, your ***driver*`open` routine should return zero to the kernel. If your ***driver*`open` routine fails to open the device, your ***driver*`open` routine should return an `errno` value to the kernel, indicating the reason the request failed.

## See also

For block devices, see "The driveropen routine" in Chapter 3.

For character devices, see "The driveropen routine" in Chapter 4.

For terminal devices, see "The open routine" in Chapter 5.

For streams devices, see "The open routine" in Chapter 6.

For streams terminal devices, see "The open routine" in Chapter 7.

## Name and purpose

print—print error messages to the user on the system console

## Synopsis

`void` ***driver*`print (`*dev,str*)

`dev_t` *dev*;

`char` **str*;

where

- *dev* is the device number.
- *str* is a pointer to a string of characters to be printed.
- *driver* is the device prefix.

## Description

Block I/O device drivers must provide a diagnostic print routine to print error messages on the console. Your driver can use the kernel's `printf` routine to output the message to the console.

## Return values

None.

## See also

"The diagnostic print routine" in Chapter 3.

`printf`(kernel) in Appendix B.

## Name

read—read data from a device

## Synopsis

`int` ***driver*`read`**`(`***dev, uio***`)`

`dev_t` ***dev***`;`

`struct uio *`***uio***`;`

where

- ***dev*** is the device number.
- ***uio*** is a pointer to the `uio` structure for the I/O request. The `uio` structure contains information about the I/O request, including the number of bytes to transfer, and a pointer to the user's buffer.
- ***driver*** is the device prefix.

## Description

The ***driver*`read`** routine of a character device driver reads data from a device when a user program issues a `read`(2) system call. The ***driver*`read`** routine is invoked with a direct pointer to the user's buffer. This allows the character device driver to buffer the data according to the needs of the device, or to directly transfer the data between the device and the user's buffer.

## Return values

If your ***driver*`read`** routine successfully reads from the device, your ***driver*`read`** routine should return zero to the kernel. If the read request fails, your ***driver*`read`** routine should return an `errno` value to the kernel, indicating the reason the request failed.

## See also

"Reading From and Writing to a Character Device" in Chapter 4.

`read`(2) in the *A/UX Programmer's Reference*.
`physio`(kernel) in Appendix B.

## Name

strategy—schedule the transfer of data between the kernel buffer cache and a device

## Synopsis

```
void driverstrategy(bp)
struct buf *bp;
```

where

- *bp* is the pointer to the `buf` structure involved in the I/O request. The `buf` structure contains information about the I/O request, including the number of bytes to transfer, the address of the kernel buffer associated with this request, and a value indicating whether data should be transferred into or out of the kernel buffer.
- *driver* is the device prefix.

## Description

The kernel calls a *driver*`strategy` routine of a block device driver to schedule the transfer of data between the buffer cache and a device.

The *driver*`read` or the *driver*`write` routine of a character device driver can invoke a *driver*`strategy` routine to transfer data directly between a device and the user's buffer.

## Return values

None.

## See also

"Performing I/O (using the strategy routine)" and "The buf structure" in Chapter 3.

"Data Transfers using physio( )" in Chapter 4.

## Name and purpose

write—write data to a device

## Synopsis

`int` ***driver*`write`**`(`***dev, uio***`)`

`dev_t` ***dev;***

`struct uio` ***uio;***

where

- ***dev*** is the device number.
- ***uio*** is a pointer to the `uio` structure for the I/O request. The `uio` structure contains information about the I/O request, including the number of bytes to transfer, and a pointer to the user's buffer.
- ***driver*** is the device prefix.

## Description

The ***driver*`write`** routine of a character device driver writes data to a device when a user program issues a `write`(2) system call. The ***driver*`write`** routine is invoked with a direct pointer to the user's buffer. This allows the character device driver to buffer the data according to the needs of the device, or to transfer the data directly between the user's buffer and the device.

## Return values

If your ***driver*`write`** routine successfully writes to the device, your ***driver*`write`** routine should return zero to the kernel. If the write request fails, your ***driver*`write`** routine should return an `errno` value to the kernel, indicating the reason the request failed.

## See also

"Reading From and Writing to a Character Device" in Chapter 4.

`write`(2) in *A/UX Programmer's Reference.*

`physio`(kernel) in Appendix B.

# Appendix B

# Kernel Routines

This appendix is a reference section for the kernel routines that a driver can call. The following routines are included in this appendix:

- `biodone()`—awakens processes waiting on the specified buffer
- `biowait()`—puts the calling process to sleep, until a corresponding call to `biodone()` is issued
- `brelse()`—returns a `buf` structure and an associated buffer to the kernel buffer cache
- `clrbuf()`—clears a buffer by filling it with zeroes
- `copyin()`—copies data from a user buffer to a driver buffer
- `copyout()`—copies data from a driver buffer to a user buffer
- `delay()`—delays execution
- `fubyte()`—copies a character from the user buffer to a driver buffer
- `fuword()`—copies an integer from the user buffer to a driver buffer
- `geteblk()`—gets a `buf` structure and associated buffer from the kernel buffer cache
- `major()`—returns the major number
- `makedev()`—creates a device number from the specified major and minor number
- `minor()`—returns the minor number
- `physio()`—performs raw I/O
- `printf()`—prints a message on the system console
- `psignal()`—sends a signal to a process
- `signal()`—sends a signal to a process group
- `sleep()`—puts a process to sleep
- `spl`*n*`()`—sets the processor interrupt level to priority level *n*
- `splx()`—resets the processor interrupt level to a previous priority level
- `subyte()`—transfers a character from a driver buffer to the user buffer
- `suword()`—transfers an integer from a driver buffer to the user buffer
- `timeout()`—sets a timer and call a specified routine when the timer expires
- `uiomove()`—moves data to and from the user buffer pointed to by the `uio` structure
- `untimeout()`—cancels a timer that was set by a previous call to `timeout()`
- `ureadc()`—writes a character to the user buffer
- `useracc()`—determines whether the driver can gain access to user address space
- `uwritec()`—reads a character from the user buffer

- `wakeup()`—wake ups processes waiting on the specified address

Entries are listed in alphabetical order and contain the following:

- the routine's name
- a synopsis of the routine's declarations and arguments
- a description of what the routine does
- the values returned from the routine
- places to look for more information

## Values and descriptions of errno

The error numbers referred to as *errno* in this appendix are listed in Table B-1, and are found in `<sys/errno.h>` and `intro(2)` of the *A/UX Programmer's Reference*. Your driver routines return *errno* error numbers to the kernel for unsuccessful requests. Also, the kernel routines that your driver can call often return zero to your driver for successful requests, and an *errno* error number for unsuccessful requests.

**Table B-1**
Kernel routine *errno* error numbers

| Number | Name | Description |
|---|---|---|
| **General error messages** | | |
| 1 | EPERM | Not superuser |
| 2 | ENOENT | No such file or directory |
| 3 | ESRCH | No such process |
| 4 | EINTR | Interrupted system call |
| 5 | EIO | I/O error |
| 6 | ENXIO | No such device or address |
| 7 | E2BIG | Argument list is too long |
| 8 | ENOEXEC | Exec format error |
| 9 | EBADF | Bad file number |
| 10 | ECHILD | No children |
| 11 | EAGAIN | No more processes |
| 12 | ENOMEM | No enough core |
| 13 | EACCES | Permission denied |

| | | |
|---|---|---|
| 14 | EFAULT | Bad address |
| 15 | ENOTBLK | Block device required |
| 16 | EBUSY | Mount device busy |
| 17 | EEXIST | File exists |
| 18 | EXDEV | Cross-device link |
| 19 | ENODEV | No such device |
| 20 | ENOTDIR | Not a directory |
| 21 | EISDIR | Is a directory |
| 22 | EINVAL | Invalid argument |
| 23 | ENFILE | File table overflow |
| 24 | EMFILE | Too many open files |
| 25 | ENOTTY | Not a typewriter |
| 26 | ETXTBSY | Text file busy |
| 27 | EFBIG | File too large |
| 28 | ENOSPC | No space left on device |
| 29 | ESPIPE | Illegal seek |
| 30 | EROFS | Read-only file system |
| 31 | EMLINK | Too many links |
| 32 | EPIPE | Broken pipe |
| 33 | EDOM | Math argument out of domain of function |
| 34 | ERANGE | Math result not representable |
| 35 | ENOMSG | No message of desired type |
| 36 | EIDRM | Identifier removed |
| 37 | ECHRNG | Channel number out of range |
| 38 | EL2NSYNC | Level 2 not synchronized |
| 39 | EL3HLT | Level 3 halted |
| 40 | EL3RST | Level 4 reset |
| 41 | ELNRNG | Link number out of range |
| 42 | EUNATCH | Protocol driver not attached |
| 43 | ENOCSI | No CSI structure available |

| | | |
|---|---|---|
| 44 | EL2HLT | Level 2 halted |
| 45 | EDEADLK | Deadlock condition |

**Network error messages**

Nonblocking and interrupt I/O

| | | |
|---|---|---|
| 55 | EWOULDBLOCK | Operation would block |
| 56 | EINPROGRESS | Operation now in progress |
| 57 | EALREADY | Operation already in progress |

Argument errors

| | | |
|---|---|---|
| 58 | ENOTSOCK | Socket operation on nonsocket |
| 59 | EDESTADDRREQ | Destination address required |
| 60 | EMSGSIZE | Message too long |
| 61 | EPROTOTYPE | Protocol wrong type for socket |
| 62 | ENOPROTOOPT | Protocol not available |
| 63 | EPROTONOSUPPORT | Protocol not supported |
| 64 | ESOCKTNOSUPPORT | Socket type not supported |
| 65 | EOPNOTSUPP | Operation not supported on socket |
| 66 | EPFNOSUPPORT | Protocol family not supported |
| 67 | EAFNOSUPPORT | Address family not supported by protocol family |
| 68 | EADDRINUSE | Address already in use |
| 69 | EADDRNOTAVAIL | Can't assign requested address |

Operational errors

| | | |
|---|---|---|
| 70 | ENETDOWN | Network is down |
| 71 | ENETUNREACH | Network is unreachable |
| 72 | ENETRESET | Network dropped connection on reset |
| 73 | ECONNABORTED | Softwawre caused connection abort |
| 74 | ECONNRESET | Connection reset by peer |
| 75 | ENOBUFS | No buffer space available |
| 76 | EISCONN | Socket is already connected |
| 77 | ENOTCONN | Socket is not connected |
| 78 | ESHUTDOWN | Can't send after socket shutdown |

| | | |
|---|---|---|
| 79 | ETOOMANYREFS | Too many references; can't splice |
| 80 | ETIMEDOUT | Connection timed out |
| 81 | ECONNREFUSED | Connection refused |
| 82 | ELOOP | Too many levels of symbolic links |
| 83 | ENAMETOOLONG | Filename too long |
| 84 | EHOSTDOWN | Host is down |
| 85 | EHOSTUNREACH | No route to host |
| 86 | ENOTEMPTY | Directory not empty |
| **Streams error messages** | | |
| 87 | ENOSTR | Not a Stream device |
| 88 | ENODATA | No data (for no delay I/O) |
| 89 | ETIME | Timer expired |
| 90 | ENOSR | Out of Streams resources |
| **Network File System error messages** | | |
| 95 | ESTALE | Stale NFS file handle |
| 96 | EREMOTE | Too many levels of remote in path |
| 97 | EPROCLIM | Too many processes |
| 98 | EUSERS | Too many users |
| 99 | EDQUOT | Disk quota exceeded |
| **Other error messages** | | |
| 100 | EDEADLOCK | Locking deadlock error |

## A summary of the kernel routines

The rest of this appendix describes the kernel routines that you can call in your driver.Entries are listed in alphabetical order and contain the following:

- the routine's name
- a synopsis of the routine's declarations and arguments
- a description of what the routine does
- the values returned from the routine
- places to look for more information

## Name

biodone—awaken processes waiting on the specified buf structure

## Synopsis

```
#include <sys/types.h>
#include <sys/buf.h>

void biodone(bp)

struct buf *bp;
```

where bp is a pointer to the buf structure associated with the buffer where I/O occurred.

## Description

The biodone routine awakens the process or processes sleeping on the buf structure. A device driver should call biodone to wake up processes put to sleep by biowait.

Note: iodone and iowait are defined in <sys/buf.h> to be equivalent to biowait and biodone.

## Return values

None.

## See also

"Kernel routines for block device drivers" in Chapter 3.

biowait(kernel)

## Name

`biowait`—put the calling process to sleep, until a corresponding call to `biowait` is issued

## Synopsis

```
#include <sys/types.h>
#include <sys/buf.h>

void biowait(bp)

struct buf *bp;
```

where *bp* is a pointer to a `buf` structure associated with the buffer where data transfer will occur.

## Description

The `biowait` routine is used by drivers that are waiting for data transfer to complete on the buffer associated with the `buf` structure. `biowait` puts the calling process to sleep on the address of the `buf` structure. The calling process is awakened by a corresponding call to `biodone` when the transfer completes.

*Note:* `iodone` and `iowait` are defined in `<sys/buf.h>` and are equivalent to `biowait` and `biodone`.

## Return values

None.

## See also

"Kernel routines for block device drivers" in Chapter 3.

`biodone`(kernel)

## Name

`brelse`—return `buf` structure and associated buffer to the kernel buffer cache

## Synopsis

```
#include <sys/types.h>
#include <sys/buf.h>

void brelse(bp)

struct buf *bp;
```

where *bp* is a pointer to the `buf` structure being returned.

## Description

`brelse` returns a `buf` structure and buffer (previously allocated by `getblk`, `geteblk`, or `bread`) to the kernel. `brelse` returns the `buf` structure to the list of free buffers and awakens any processes on that list that might be sleeping.

## Return values

None.

## See also

"Kernel routines for block device drivers" in Chapter 3.

`geteblk`(kernel)

## Name

`clrbuf`—clear buffer

## Synopsis

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

void clrbuf(bp)

struct buf *bp;
```

where *bp* is the pointer to the buf structure.

## Description

The `clrbuf` macro (defined in `<sys/buf.h>`) zeros the indicated buffer and sets the `b_resid` field of the `buf` structure to 0.

## Return values

None.

## See also

"The buf structure" in Chapter 3.

## Name

`copyin`—copy data from user buffer to driver buffer

## Synopsis

```
int copyin (userbuf, driverbuf, n)
char *driverbuf, *userbuf;
int n;
```

where

- *userbuf* is the address of the user buffer
- *driverbuf* is the address of the driver buffer
- *n* is the number of bytes to copy

## Description

`copyin` copies data from a user buffer to a driver buffer.

**Return values**

| Value | Meaning |
|---|---|
| 0 | Success |
| *errno* | Failure |

## See also

"Utility Routines and Macros" in Chapter 2.

`copyout`(kernel)

`subyte`(kernel)

`suword`(kernel)

`fubyte`(kernel)

`fuword`(kernel)

## Name and purpose

`copyout`—copy data from driver buffer to user buffer

## Synopsis

`int copyout(`***driverbuf, userbuf, n***`)`

`char *`***driverbuf***`, *`***userbuf***`;`

`unsigned n;`

where

- ***driverbuf*** is the address of the driver buffer.
- ***userbuf*** is the address of the user buffer.
- ***n*** is the number of bytes to copy.

## Description

`copyout` copies data from a driver buffer to a user buffer.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| *errno* | Failure |

## See Also

"Reading from and writing to a user buffer" in Chapter 2.

`copyout`(kernel)

`subyte`(kernel)

`suword`(kernel)

`fubyte`(kernel)

fuword(kernel)

## Name

`delay`—delay execution

## Synopsis

```
void delay(ticks)
int ticks;
```

where *ticks* is the number of clock cycles to delay (the variable `v.v_hz` contains the number of clock cycles per second).

## Description

`delay` makes a process wait for a specific time interval before resuming execution. `delay` puts the user process to sleep, so your driver must not call `delay` from within an interrupt routine.

## Return values

None.

## See also

"Delaying Execution" in Chapter 2.

## Name

`fubyte`—copy a character from the user buffer to a driver buffer

## Synopsis

```
int fubyte (userbuf)
char *userbuf;
```

where *userbuf* is the address of the user buffer.

## Description

`fubyte` copies a single character from the user buffer to the driver buffer.

## Return values

| Value | Meaning |
|---|---|
| 0-255 | The ASCII value of the character successfully returned. |
| –1 | Failure |

## See also

"Reading from and writing to a user buffer" in Chapter 2.

`fuword`(kernel)

`subyte`(kernel)

`suword`(kernel)

`copyin`(kernel)

`copyout`(kernel)

## Name

`fuword`—copy integer from the user buffer to a driver buffer

## Synopsis

`int fuword (`*`userbuf`*`)`

`int *`*`userbuf`*`;`

where *userbuf* is the address of the user buffer.

## Description

`fuword` copies an integer from a user buffer to a driver buffer.

## Return values

| Value | Meaning |
|---|---|
| –1 | Failure (see note below) |
| Any other value | Success |

*Note*: –1 is also returned when fuword fetches a 0xFFFFFFFF from memory, even when no error condition exists.

## See also

"Reading from and writing to a user buffer" in Chapter 2.

`fubyte`(kernel)

`subyte`(kernel)

`suword`(kernel)

`copyin`(kernel)

`copyout`(kernel)

## Name

geteblk—get a buf structure and associated buffer from the kernel buffer cache.

## Synopsis

```
#include <sys/types.h>
#include <sys/buf.h>

struct buf* geteblk(size)

        int size;
```

where *size* is the size of the buffer.

## Description

geteblk retrieves a buf structure and associated buffer of size bytes from the buffer cache. geteblk returns a pointer to the buf structure to the calling routine. If no buf structures are available, geteblk sleeps until one becomes available. Thus, your driver must not call geteblk from within an interrupt handler.

When the device driver strategy routine receives a buffer header from the kernel the necessary fields are already initialized. However, when a device driver calls geteblk to allocate buffers, the device driver must set up some of the fields of the buffer header before calling the strategy routine.

Important fields in the buffer header are as follows:

- b_flags contains bits that indicates the status of the buffer (B_BUSY flag) and tells the driver whether the device is to be read from or written to (B_READ or B_WRITE flag).
- av_forw and av_back are a pair of pointers that maintain a doubly linked list of "free" blocks (blocks that can be reallocated for another transaction). A driver can use these lists to link the buffer into driver worklists.
- b_bcount is a count of the number of bytes to be transferred to or from the buffer.
- b_error holds the error code to be assigned by the kernel to the u_error field of the user data structure. It is set in conjunction with the B_ERROR flag after an I/O operation.
- b_dev holds the device number of the device being accessed. The high-order 8 bits contain the major number and the low-order 8 bits contain the minor number.

- `b_un.b_addr` is the virtual address of the buffer controlled by the buff structure.Data is read from or written to this address to/from the device.
- `b_blkno` is the device block to be accessed (the minor number determines this device).
- `b_resid` is the number of bytes not transferred if error has occurred.
- `b_start` is the start time of the I/O; it measures device response time.

The only fields that a driver may change are `b_flags`, `av_forw`, `av_back`, `b_error`, and `b_resid`.

The following list describes the states of some of the fields when `geteblk` receives them and how they must be initialized.

- `b_flags`—The `B_BUSY` flag in this field is set to indicate that the buffer is in use. The driver must set the `B_READ` or `B_WRITE` flag to indicate the type of transfer being done.
- `b_bcount`—This field is set to the number of bytes in the buffer.
- `b_blkno`—`geteblk` doesn't initialize this field; thus, it must be initialized by your driver.

The remaining fields in the buffer header can be used unchanged.

## Return values

`geteblk` returns a pointer to a `buf` structure that the driver can use. Your driver should call the `brelse` routine to return the buffer to the kernel.

## See also

"Kernel Routines for block device drivers" in Chapter 3.

"Data transfers using uiomove( )" in Chapter 4.

`uiomove`(kernel)

## Name

`major`—return major number.

## Synopsis

```
#include <sys/types.h>
#include <sys/sysmacros.h>

int major(dev)
```

where *dev* is the device number.

## Description

The `major` macro (from `sysmacros.h`) returns the major number when passed the external device number.

## Return values

`major` returns the high-order 8 bits of the device number.

## See also

"Device Files" in Chapter 2.

makedev(kernel)

minor(kernel)

## Name

`makedev`—encode major and minor number.

## Synopsis

`int makedev (`*x*,*y*`)`

where

- *x* is the major number.
- *y* is the minor number.

## Description

The `makedev` macro (from `sysmacros.h`) encodes the major and minor numbers to create the external device number.

## Return values

When supplied the major and minor numbers, `makedev` returns the 16-bit device number.

## See also

"Device Files" in Chapter 2.

major(kernel)

minor(kernel)

## Name

`minor`—return minor number.

## Synopsis

```
#include <sys/types.h>
#include <sys/sysmacros.h>
```

`int minor(`*dev*`)`

where *dev* is the device number.

## Description

The `minor` macro (from `sysmacros.h`) returns the minor number when passed the external device number.

## Return values

`minor` returns the low-order 8 bits of the device number.

## See also

"Device Files" in Chapter 2.

major(kernel)

makedev(kernel)

## Name and purpose

physio—perform raw I/O

## Synopsis

```
#include <sys/types.h>
#include <sys/buf.h>
#include <sys/uio.h>

int physio(strat, bp, dev, rw, uio)

int (*strat) ();

struct buf *bp;

dev_t dev;

int rw;

struct uio *uio;
```

where

- *strat* is the address of the *driver*strategy routine.
- *bp* is a pointer to a buf structure.
- *dev* is a device number that is received as an argument from the *driver*read or *driver*write routine.
- *rw* is a flag that indicates whether the operation is a read or write.
- *uio* is a pointer to the uio structure associated with this request.

## Description

physio sets up a buf structure describing the I/O request. For example, physio fills in b_bcount with the number of bytes to transfer, sets B_READ or B_WRITE in the b_flags field to indicate the direction to transfer data, and sets b_un.b_addr to point to the user's buffer. physio then locks the user process in memory and calls the *driver*strategy routine, passing a pointer to the buf structure as a parameter. When the *driver*strategy routine returns, physio waits for the I/O request to complete by calling biowait. When the transfer completes, the driver interrupt routine awakens the user process by calling biodone. physio then updates information in the uio structure and returns to the *driver*read or *driver*write routine.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| *errno* | Failure |

## See also

"Data Transfers using physio( )" in Chapter 4.

biodone(kernel)

## Name

`printf`—print message on the system console

## Synopsis

`void printf (`*format*`[ , `*arg*`...])`

`char *`*format*`;`

where

- *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching zero or more *args*.
- *arg* is an argument to be converted and output.

## Description

The kernel `printf` routine prints characters to the console. Note that this is the kernel's `printf` routine, not the C library `printf` routine, although the two routines are very similar. `printf`(kernel) supports the following limited subset of `printf`(3f) conversion specifications:

| | |
|---|---|
| %d,%o,%u,%x | The integer arg is converted to signed decimal (d), unsigned octal (o), decimal (u), or hexadecimal(x) notation. |
| %c | The character arg is printed. |
| %s | The arg is taken to be a string (character pointer) and characters from the string are printed until a NULL character (\0) is encountered or the number of characters indicated by the precision specification is reached. |

## Return value

None.

## See also

"The diagnostic print routine" in Chapter 3.

print(driver) in Appendix A.

printf(3S)

## Name and purpose

`psignal`—send a signal to a process

## Synopsis

```
#include <sys/types.h>
#include <sys/proc.h>
#include <sys/signal.h>
#include <sys/time.h>
#include <sys/resource.h>

void psignal (proc, sig)

struct proc *proc;

int sig;
```

where

- *proc* is the pointer to the `proc` structure entry for the process.
- *sig* is the signal itself.

## Description

`psignal` sends a signal to a particular process. The routine does this by marking in the `proc` structure that the process should receive a signal and then enabling the job to run.

## Return values

None.

## See also

"Process context and the user structure" in Chapter 2.

"Sending a signal to a user process" in Chapter 2.

signal(2) in *A/UX Programmer's Reference.*

## Name

`signal`—send a signal to a process group

## Synopsis

```
#include <sys/types.h>
#include <sys/proc.h>
#include <sys/signal.h>
#include <sys/time.h>
#include <sys/resource.h>

void signal (pgrp, sig)

int pgrp;

int sig;
```

where

- *pgrp* is is the process group which will be sent a signal.
- *sig* is the signal itself (see `signal`(2) in the *A/UX Programmer's Reference* for integer values).

## Description

The kernel `signal` routine sends a signal to a specified process group. Do not confuse this routine with `signal`(2), which specifies how the calling process handles signals that are received.

## Return values

None.

## See also

"Sending a signal to a user process" in Chapter 2.

psignal(kernel)

## Name and purpose

`sleep`—put a user process to sleep

## Synopsis

```
#include <sys/param.h>
int sleep (event, priority)
caddr_t event;
int priority;
```

where

- *event* is the address of some data structure used by the driver.
- *priority* is the priority level.

## Description

`sleep` makes a process wait until a certain event occurs. To the `sleep` routine, an event is an address that the sleeping process and the `wakeup` routine synchronize on. Other processes can run while a process is sleeping. The kernel marks the process state "asleep," saves the sleep event and priority, and puts it into a hashed queue of sleeping processes.

Sleep priorities range from 0 to 127, 0 having the highest priority and 127 the lowest priority. Processes sleeping at a priority less than the parameter `PZERO` can't be interrupted by signals, although they can be swapped out. For this reason, it is not a good idea to sleep with priority less than `PZERO` on an event that might never occur. In general, sleeps at less than `PZERO` should only be made for fast events such as disk and tape I/O.

*Caution:* Never call sleep in an interrupt routine, becausethe current process is probably not the one that should go to sleep.

## Return Values

The PCATCH bit of *priority* is OR'ed into the priority field of the proc structure when a driver wants any signals that occur during sleep to be ignored and handled later (for example, page faults and Streams processing). `sleep` returns the following values if PCATCH is set:

| Value | Meaning |
|---|---|
| 0 | No signal occurred |
| 1 | Signal occurred |

If PCATCH is not set, the return value to `sleep` has no meaning (0 is returned).

## Examples

The first example shows many processes that are competing for a resource:

```
#define     X_LOCK      1
#define     X_WANT      2
int x_flag;
X_LOCK()
{     while
            (x_flag) {
            x_flag |=X_WANT;
            sleep(&x_flag,PZERO);
            }
            x_flag = X_LOCK;
}
x_unlock()
{
      if(x_flag&X_WANT)
            wakeup(&x_lock);
      x_lock = 0;
}
```

The second example shows synchronization using an interrupt:

```
int x_flag;
x_wait()
{
        spl6();
        x_flag = 1;
        while (x_flag) {
                sleep(&x_flag,PZERO)
        }
        sp0();
        .
        .
        .
x_intr()
{
        if (x_flag)
                wakeup(&x_flag);
        x_flag = 0;
}
```

## See also

"Notifying a process of I/O completion" in Chapter 2.

"Waiting for I/O to complete" in Chapter 2.

wakeup(kernel)

## Name

`spl`*n*—set processor interrupt level

## Synopsis

```
short int splnO
```

where *n* is the priority level (0-7), with 0 having the lowest priority and 7 the highest priority.

## Description

`spl`*n* enables interrupts having priority levels greater than *n*. This routine prevents unwanted interrupts from reaching a device.

Important values of *n* for the Macintosh II are as follows:

| *n* | Description |
|---|---|
| 7 | Interrupt switch |
| 6 | Power-on switch |
| 4 | On-board SCCs |
| 2 | Slots, SCSI disk |
| 1 | Clock, ADB |
| 0 | System running |

For example, spl2 disables priority levels 2, 1, and 0.

## Return values

`spl`*n* returns the contents of the status register before the routine was called.

## See also

"Setting Processor Levels" in Chapter 2.

## Name

`splx`—reset processor interrupt priority level

## Synopsis

void splx(*s*)

int *s*;

where *s* is the value of the status register returned by the previous spl*n* call.

## Description

`splx` sets the interrupt priority level back to its previous state (before `spl`*n* was called).

## Return values

None.

## See also

"Setting Processor Levels" in Chapter 2.

spl*n*(kernel)

## Name

subyte—transfer a character from a driver buffer to a user buffer

## Synopsis

```
int subyte (userbuf, c)
char *userbuf, c;
```

where

- *userbuf* is the address of the user buffer
- *c* is a character to copy

## Description

subyte transfers a character from a driver buffer to a user buffer.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## See also

"Reading From and Writing To a User Buffer" in Chapter 2.

suword(kernel)

fuword(kernel)

fubyte(kernel)

copyin(kernel)

copyout(kernel)

## Name

suword—transfer an integer from a driver buffer to a user buffer

## Synopsis

```
int suword (userbuf, i)
char *userbuf,
int i;
```

where

- *userbuf* is the address of the user buffer.
- *i* is the integer to be copied.

## Description

suword transfers an integer from a driver buffer to a user buffer.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## See also

"Reading From and Writing To a User Buffer" in Chapter 2.

subyte(kernel)

fuword(kernel)

fubyte(kernel)

copyin(kernel)

## Name

`timeout`—set a timer and when the timer expires call the specified routine

## Synopsis

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

void timeout (func, arg, interval)

int (*func) ();

caddr_t arg;

int interval;
```

where

- *func* is the routine you want to call after the given interval. The specified *func* is called at clock interrupt time, so the routine called by timeout must not call sleep.
- *arg* is the argument to the function.
- *interval* is the time number of clock ticks to delay before calling *func*. This value is limited to ((2**31)–1), since it must appear to be positive and since only a bounded number of timeouts can be occurring at any time.

## Description

`timeout` sets a timer and calls the specified routine when the timer expires. This can be useful when you want a timeout to occur and control to jump to another routine after a given time period.

Timeouts are only guaranteed to happen after the time specified. This means that they may occur some time after the interval has expired. Call `untimeout`(kernel) to cancel a previous timeout request.

## Return values

None.

## See also

"Setting a timeout" in Chapter 2.

"Removing a timeout" in Chapter 2.

untimeout(kernel)

## Name

uiomove—move data to and from the user's buffer specified by the uio structure

## Synopsis

```
#include <sys/types.h>
#include <sys/uio.h>
```

`int uiomove` *(address,byte_count, flag,*uio)*

`caddr_t` *address;*

`int` *byte_count;*

`int` *flag;*

`struct uio` **uio;*

where

- *address* is the address of the buffer where data transfer will occur.
- *byte_count* is the number of bytes to transfer.
- *flag* is either `UIO_READ` or `UIO_WRITE`.
- *uio* is the uio structure involved.

## Description

uiomove moves data between an area described by a `uio` structure and a kernel address. The `uio` structure is updated automatically.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| *errno* | Failure |

## See also

"Data transfers using uiomove" in Chapter 2.

geteblk(kernel)

## Name

`untimeout`—cancel timeout

## Synopsis

```
#include <sys/types.h>
void untimeout (func, arg)
int (*func) ();
caddr_t arg;
```

where

- *func* is the routine your driver calls after the given time interval.
- *arg* is the argument to that routine.

## Description

`untimeout` cancels a previous timeout request. Call `untimeout` after the event awaited has happened. This will prevent the process from timing out and jumping to *func*.

## Return values

None.

## See also

"Setting a timeout" in Chapter 2.

"Removing a timeout" in Chapter 2.

timeout(kernel)

## Name and purpose

ureadc—deliver a character to user's buffer

## Synopsis

```
int ureadc (c, uio)
char c;
struct uio *uio;
```

where

- *c* is the character delivered.
- *uio* is the uio structure for the operation.

## Description

ureadc delivers a character to a user's buffer when a read(2) system call is made.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| *errno* | Failure |

## See also

"Reading From and Writing To a User Buffer" in Chapter 2.

uwritec(kernel)

useracc(kernel)

## Name

useracc—determine whether driver can access user address space

## Synopsis

```
int useracc (addr, count, access)
caddr_t addr;
int count;
int access;
```

where

- *addr* is the address to be accessed.
- *count* is the number of bytes to be accessed.
- *access* is the type of access. It can be either B_READ (read access) or B_WRITE (write access).

## Description

useracc determines whether your driver can gain access to a specified user address space. This routine must be called in user context.

## Return values

| Value | Meaning |
|---|---|
| 1 | Can access address space |
| 0 | Can't access address space |

## See also

"Reading From and Writing To a User Buffer" in Chapter 2.

ureadc(kernel)

uwritec(kernel)

## Name and purpose

`uwritec`—retrieve character from user buffer

## Synopsis

`int uwritec(`*uio*`)`

where *uio* is the uio structure for the I/O operation.

## Description

`uwritec` retrieves a character placed in a user's buffer by a `write`(2) system call.

## Return values

| Value | Meaning |
|---|---|
| 0-255 | Success—the ASCII value of the character retrieved is returned. |
| –1 | Error |

## See also

"Reading and Writing To and From a User Buffer" in Chapter 2.

write(2) in *A/UX Programmer's Reference*.

ureadc(kernel)

useracc(kernel)

## Name

wakeup—wake up a sleeping process

## Synopsis

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

void wakeup (event)

caddr_t event;
```

where *event* is the address of the data structure used by the driver in a previous call to `sleep`.

## Description

`wakeup` awakens all user processes sleeping on event. wakeup changes the process state from "asleep" to "ready to run." Sleeping processes (those that are marked "asleep") are removed from the sleeping processes queue, and are placed on a list of processes that are able to run.

## Return values

None.

## See also

"Notifying a process of I/O completion" in Chapter 2.

"Waiting for I/O to complete" in Chapter 2.

sleep(kernel)

# Appendix C

# Slot Library Routines

This appendix describes Slot ROM Library routines that a driver can call. The Slot ROM Library is designed to provide a simple interface to on-board ROM resources for each of the six expansion slots on the Macintosh II. No knowledge of the ROM format or board addressing requirements is presumed. Before you use this library, you should be familiar with slot data structures for the Macintosh II. For more information about them, see *Developing Cards and Drivers for Macintosh II and Macintosh SE.*

The Slot Library contains three types of routines: user, utility, and low-level. User routines can be called from user or kernel routines. Utility routines are used to gain access to slot ROM data structures, other resources, or other user programs. Low-level routines perform ROM access operations and operating-system-specific functions. For a summary of all routines in the library, see `slots(3x)` in the *A/UX Programmer's Reference.*

## User routines

User routines perform simple functions such as reading information from a slot ROM and filling in slot data structures. The first parameter to user routines is the board's slot number, which can be one of the following:

- the NuBus slot number (9 to 14)
- the physical ROM base address of the slot ROM
- the virtual base address of the slot ROM

If the program that calls the library routine is part of the A/UX operating system, all of the system's resources are directly available to the program by using a physical address. Slot ROM physical addresses are hexadecimal values having the following format:

`0xFs0FF0000`

where *s* is the NuBus slot number of the board containing the ROM.

Optional parameters to user routines are a pointer to a buffer and the length of the buffer. User routines that have only one parameter return their results directly.

User library functions search the resource list from the resource directory for a desired resource (for example, the board resource list is searched for the board ID). To read the board ID, the `RBL_BOARDID` type is located in the resource list for the board resource, and the 16 bits of board ID is read into a user data structure.

## Utility routines

Utility routines handle the access to slot ROM structures. The utility routine `slot_directory` reads the resource directory into a buffer. Other utility routines call `slot_directory` to read the resource directory from ROM. When this directory is read into contiguous RAM, the calling routine locates the requested type of resource in the resource directory (for example, the board or the Ethernet resources).

Although the A/UX kernel most commonly uses the utility routines to search for a board resource, `slot_data` and `slot_resource` can be used to gain access to resources for other than boards and other user programs. Doing this allows you to read other vendor resource types from ROM.

*Note:* Vendor resource types reserved for use by Apple Computer are values between 1 and 127 (decimal); other vendor-defined types may be any value between 128 and 255 (decimal).

To read a subresource of the board resource list, the slot ROM user functions call `slot_resource` or `slot_data` with the ***kind*** parameter set to `RD_BOARD` (which is defined as 1) and the request parameter set to the desired resource type (for example, `RBL_BOARDID`). The only user routine that doesn't search for the board resource is the user routine `slot_ether_addr`, which requests the Ethernet resource type `RD_ETHER`.

## Low-level routines

Low-level routines read data from ROM and call operating-system-specific functions. `slot_rom_data` performs the actual read. `slot_check_crc` checks the ROM checksum and calls `slot_bytelane` to determine what slot address lines are being used by hardware. The `slot_seg_violation`, `slot_catch`, and `slot_ignore` routines are operating system specific and are used to detect ROM read failures.

The library routines found in this appendix are listed next.

## User routines

User routines perform simple functions such as reading information from a slot ROM and filling in slot data structures. The user routines are listed here:

```
slot_PRAM_init
slot_board_flags
slot_board_id
slot_board_name
slot_board_type
slot_ether_addr
slot_primary_init
slot_part_num
slot_rev_level
slot_serial_number
slot_vendor_id
```

## Utility routines

Utility routines handle the access to slot ROM structures and are listed here:

```
slot_byte
slot_data
slot_directory
slot_long
slot_resource
slot_resource_list
slot_structure
slot_word
```

## Low-level routines

Low-level routines read data from ROM and call operating-system-specific functions, and are listed here:

```
slot_seg_violation
```

```
slot_catch
slot_ignore
slot_address
slot_byte lane
slot_calc_pointer
slot_rom_data
slot_check_crc
slot_header
```

## Name

`slot_PRAM_init`

## Synopsis

```
int slot_PRAM_init (slot, pp)
int slot;
struct PRAM *pp;
```

where

- *slot* is a NuBus slot number or ROM base address for the board ROM.
- *pp* is the address of a PRAM structure to be filled in by `slot_PRAM_init`.

## Description

`slot_PRAM_init` fills in the `PRAM` structure referenced by the parameter *pp* from slot ROM for the board located in *slot*.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## Example

```
int slot = 9;
struct PRAM pram;
if(slot_PRAM_init(slot, &pram) != 0) {
        /* error ... */
```

```
}
else {
        /* no problem, pram is now useable */
}
```

## Name

`slot_board_flags`

## Synopsis

`u16 slot_board_flags(`*slot*`)`

`int` *slot*;

where *slot* is a NuBus slot number or a ROM base address for the board ROM.

## Description

`slot_board_flags` reads and returns the board flags for the board located in *slot*. Board flag bit definitions are found in `<slots.h>`.

## Return values

| Value | Meaning |
|---|---|
| *board flag bits* | Success |
| 0xFFFF | Failure |

## Example

```
u16 boardflags;

int slot;

if((boardflags = slot_board_flags(slot)) == 0xFFFF) {

        /* error ... */

}
```

## Name

`slot_board_id`

## Synopsis

```
u16 slot_board_id(slot)
        int slot;
```

where *slot* is a NuBus slot number or a ROM base address for the board ROM.

## Description

`slot_board_id` returns the unique board number (assigned by Apple Computer) for the board found in *slot*.

## Return values

| Value | Meaning |
|---|---|
| *board ID* | Success |
| `0xFFFF` | Failure |

## Example

```
u16 boardid;

int slot;

if((boardid = slot_board_id(slot)) == 0xFFFF) {
 /* error ... */
}
```

## Name

`slot_board_name`

## Synopsis

`int slot_board_name(`*slot, data, size*`)`

```
int slot;
char *data;
int size;
```

where

- *slot* is a NuBus slot number or ROM base address for the board ROM.
- *data* is a pointer to a character buffer to hold the board name string.
- *size* is the number of characters that can be stored in the buffer pointed to by data.

## Description

`slot_board_name` reads the board name string from ROM located on the board in *slot*. This routine fails if *slot* is not a valid slot number.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## Example

```
char string[80];

int slot = 9;

if(slot_board_name(slot, &string, sizeof(string)) == -1) {
```

```
        /* error ... */
}
```

## Name

`slot_board_type`

## Synopsis

`int slot_board_type(`*slot, data*`)`

```
int slot;
char *data;
```

where *slot* is a NuBus slot number or a ROM base address for the board ROM.

## Description

`slot_board_type` returns the unique board type, an unsigned 64-bit (8-byte) quantity for the board found in *slot*. The board type is a board class, such as network or memory.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success (data is valid). |
| –1 | Failure |

## Example

```
char boardtype[8];
int slot;
if(slot_board_type(slot, boardtype) == -1) {
        /* error ... */
}
```

## Name

`slot_ether_addr`

## Synopsis

```
int slot_ether_addr(slot, string)
        int slot;
        char *string;
```

where

- *slot* is a NuBus slot number or a ROM base address for the board ROM.
- *string* is a pointer to a character buffer that holds six bytes of Ethernet address (which by definition is six bytes long).

## Description

`slot_ether_addr` reads the Ethernet address out of ROM (six bytes) on the board located in *slot*. `slot_ether_addr` fails if *slot* is not a valid slot number, if the board is not an Ethernet interface or if there is a ROM error.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## Example

```
char addr[6];
int slot = 9;
if(slot_ether_addr(slot, &addr) == -1) {
        /* error ... */
```

## Name

`slot_primary_init`

## Synopsis

```
int slot_primary_init(slot, pp)
        int slot;
        struct prim *pp;
```

where

- *slot* is a NuBus slot number or a ROM base address for the board ROM.
- *pp* is the address of a primary structure to be filled in by `slot_primary_init`.

## Description

`slot_primary_init` fills in the primary structure pointed to by the parameter *pp* from the board's slot ROM that is located in *slot*.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success (the ROM read for *slot* succeeds and *pp* is left pointing to a valid primary structure. |
| –1 | Failure |

## Example

```
int slot = 9;

struct prim primary;

if(slot_primary_init(slot, &primary) != 0) {
        /* error ... */
```

```
}
else {
        /* no problem, primary is now useable */
}
```

## Name

`slot_part_num`

## Synopsis

`int slot_part_num(`*slot, data, size*`)`

```
int slot;
char *data;
int size;
```

where

- *slot* is a NuBus slot number or a ROM base address for the board ROM.
- *data* is a pointer to a character buffer that holds the board part number string.
- *size* is the number of characters that can be stored in the buffer pointed to by *data*.

## Description

`slot_part_num` reads the board part number string out of ROM for the board located in *slot*. This routine fails if *slot* is not a valid slot number.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## Example

```
char string[80];

int slot = 9;

if(slot_part_num(slot, &string, sizeof(string)) == -1) {
```

```
        /* error ... */
}
```

## Name

`slot_rev_level`

## Synopsis

```
int slot_rev_level(slot, data, size)
        int slot;
        char *data;
        int size;
```

where

- *slot* is a NuBus slot number or ROM base address for the board ROM.
- *data* is a pointer to a character buffer to hold the board revision level string.
- *size* is the number of characters that can be stored in the buffer pointed to by *data*.

## Description

`slot_rev_level` reads the board revision level string from ROM for the board located in *slot*. This routine fails if *slot* is not a valid slot number.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## Example

```
char string[80];

int slot = 9;

if(slot_rev_level(slot, &string, sizeof(string)) == -1) {
```

```
 /* error ... */
}
```

## Name

`slot_serial_number`

## Synopsis

`int slot_serial_number(`*slot, data, size*`)`

```
int slot;
char *data;
int size;
```

where

- *slot* is a NuBus slot number or ROM base address for the board ROM.
- *data* is a pointer to a character buffer to hold the board serial number string.
- *size* is the number of characters that can be stored in the buffer pointed to by *data*.

## Description

`slot_serial_number` reads the board serial number string from ROM for the board located in *slot*. `slot_serial_number` fails if *slot* is not a valid slot number.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## Example

```
char string[80];

int slot = 9;

if(slot_serial_number(slot, &string, sizeof(string)) == -1) {
```

```
        /* error ... */
    }
```

## Name

`slot_vendor_id`

## Synopsis

`int slot_vendor_id(`*slot*, *data*, *size*`)`

`int` *slot*;

`char` **data*;

`int` *size*;

where

- *slot* is a NuBus slot number or ROM base address for the board ROM.
- *data* is a pointer to a character buffer that holds the board vendor identification string.
- *size* is the number of characters that can be stored in the buffer referenced by *data*.

## Description

`slot_vendor_id` reads the board vendor identification string from ROM for the board located in *slot*. `slot_vendor_id` fails if *slot* is not a valid slot number.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## Example

```
char string[80];

int slot = 9;
```

```
if(slot_vendor_id(slot, &string, sizeof(string)) == -1) {
/* error ... */
}
```

## Name

`slot_board_vendor_info`

## Synopsis

`int slot_board_vendor_info(`***kind, slot, data, size***`)`

`int` ***kind***;

`int` ***slot***;

`char *`***data***;

`int` ***size***;

where

- ***kind*** is the type of vendor information to be read out of the ROM.
- ***slot*** is a NuBus slot number or ROM base address for the board ROM.
- ***data*** is a pointer to a data buffer to hold the ROM data.
- ***size*** is the number of bytes available in the buffer pointed to by ***data***.

## Description

`slot_board_vendor_info` is a vendor information structure access routine. This structure contains a list of pointers to strings in ROM that contain vendor information such as the vendor ID, the board revision level, and the board serial and part number strings. All of the user routines call `slot_board_vendor_info` and pass in the appropriate ***kind*** constant for their function. For example, the user routine `slot_part_num` passes the constant `B_PN` (board part number) to `slot_board_vendor_info` and expects a maximum of ***size*** bytes of the board part-number string to be returned in the buffer pointed to by ***data***.

`slot_board_vendor_info` calls the utility routine `slot_resource` to read the vendor list structure out of the board resource directory. If the call to `slot_resource` fails, an error is returned to the caller immediately.

After `slot_resource` successfully reads the vendor information structure, `slot_board_vendor_info` searches the list for the requested type of vendor information. If found in the list, the information string is copied into the user buffer (up to *size* bytes), by calling `slot_structure` and returning the status of that call. If not found in the list, an error value is returned to the calling program.

## Return values

| Value | Meaning |
|---|---|
| *n* | Success (the number of bytes of information copied into the user buffer is returned. |
| –1 | Failure |

## Example

```
#include <slots.h>

char string[80];

int slot = 9;

if(slot_board_vendor_info(B_PN, slot, &string, sizeof(string)) == -1) {
 /* error ... */
}
```

## Name

`slot_byte`

### Synopsis

`char slot_byte(`*rbp*`)`

`struct rsrc_byte` *rbp*;

where *rbp* is a structure that defines a 32-bit quantity in ROM.

## Description

`slot_byte` returns the least significant byte of a 32-bit quantity contained in ROM that is part of the `rsrc_byte` structure (defined in `slots.h`). `slot_byte` is used when the resource type of data stored in ROM is a naked byte. This routine, the `slot_long`, the `slot_word`, and the `slot_structure` routines are used to access the four types of low-level ROM data types.

## Return values

| Value | Meaning |
|---|---|
| *char* | Success (the 8-bit character stored in *rbp* is returned). |
| None | All 8-bit values are legal. |

## Example

```
struct rsrc_byte rbp;

char c;

char *address;

int j;

struct rom_idoffset rd[20];

        /*

        * Create a ROM base address.
```

```
        */
if((address = slot_address(i)) < 0) {
        /* Error... */
}
        /*
        * Get the resource directory from ROM.
        */
if(slot_directory(address, rd, 20) < 0) {
        /* Error... */
}
        /*
        * Find a resource of type BYTE, and read the value into
        * variable "c."
        */
for(j = 0; ((j < 20) && (rd[j].r_id != RD_EOLIST)); j++) {
        if( rd[j].r_id == RD_BYTE) {
                rbp = (struct rsrc_byte)rd[j];
        c = slot_byte(rbp);
        break;
        }
}
```

## Name

`slot_data`

## Synopsis

`int slot_data(`***slot, kind, request, datap, size***`)`

`int` ***slot;***

`int` ***kind;***

`int` ***request;***

`int` ****datap;***

`int` ***size;***

where

- ***slot*** is a NuBus slot number or a ROM base address for the board ROM.
- ***kind*** is the slot resource list type.
- ***request*** is the resource type to be accessed.
- ***data*** is a pointer to a data word that will hold the ROM data (if found).
- ***size*** is the number of bits to be stored into the word pointed to by ***data***.

## Description

`slot_data` reads ***size*** bits of data and places the information into a data word pointed to by the ***datap*** structure. The data is stored in a substructure of a resource list having type ***kind***. The data itself is a resource of type ***request***. The resource is a value up to 32 bits wide, so it can be a byte, a word (16 bits), or a long (32 bits).

Several other slot library routine call the low-level `slot_data` access routine. `slot_data` passes the parameter ***slot*** to the library routine `slot_address` to create the ROM base address from the slot number. The base address is then passed as one of the parameters to `slot_resource_list` to read the resource list of type ***kind***. `slot_data` scans the resource list returned by the previous call for the resource of type ***request***. When the resource is found, the ***size*** parameter determines which of the three possible data access routines will place the data into the user data word. An error return is immediately sent to the user if any of the routines called in `slot_data` return an error.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success |
| –1 | Failure |

## Example

```
int slot = 9;
int data;
/*
* Read the board ID resource from the board resource list for
* the board in slot 9. The board ID is 16 bits wide.
*/
if(slot_data(slot, RD_BOARD, RBL_BOARDID, &data, 16) == -1) {
/* error ... */
}
```

## Name

`slot_directory`

## Synopsis

`int slot_directory(`*slot, data, size*`)`

`int` *slot*;

`char *`*data*;

`int` *size*;

where

- *slot* is a NuBus slot number or a ROM base address for the board ROM.
- *data* is a pointer to a buffer to hold the resource directory.
- *size* is the number of `rom_idoffset` structures to be placed in the buffer pointed to by *data*.

## Description

`slot_directory` reads *size* entries of the resource directory for *slot* into the buffer pointed to by *data*. The resource directory is a structure containing all of the resources supported by the board for which the ROM was created. The resource directory is where all the searches for board resources in ROM begin. Each entry in the resource directory is a `rom_idoffset` structure. This structure, defined in `slots.h`, consists of two fields: an 8-bit ID and a 24-bit offset.

The `slot_directory` routine uses `slot_header` to read the ROM header and create a ROM base address from the *slot* parameter. The base address and the resource directory

offset from the ROM header are passed to `slot_calc_pointer` which creates a pointer to the resource directory in ROM. Using this pointer, the low-level library function `slot_rom_data` is called to read *size* entries of the resource directory into the buffer pointed to by *data*.

## ırn values

| ıue | Meaning |
|---|---|
| ) | Success |
| –1 | Failure |

## Example

```
int i, j;
unsigned address;
struct rom_idoffset rd[20];
/*
* Loop through all of the NuBus slots.
*/
for(i = 9; i < 14; i++) {
/*
* Read and print the resource directories for all the slots.
* First print a header
*/
printf("Resource Directory for slot %d:0, i);
/*
* Create a ROM base address.
*/
if(address = slot_address(i) < 0) {
/*
* This shouldn't happen since we are passing in valid slot
* numbers.
*/
 exit(1);
}
if(slot_directory(address, rd, 20*sizeof(struct rom_idoffset)) < 1) {
printf("No directory found0);
```

```
continue;
}
for(j = 0; ((j < 20) && (rd[j].r_id != RD_EOLIST)); j++) {
/*
* Print the contents of the directory up to the End Of List
* marker (or the maximum list size of 20)
*/
 printf(" %d %x %X0, j,
}
}
```

## Name

`slot_long`

## Synopsis

`u32 slot_long(`***address***`, `***rdp***`)`

`struct rom_idoffset` ***rdp***;

`unsigned` ***address***;

where

- ***rdp*** is a structure defining a 32-bit quantity in ROM that contains a pointer to the long data to be read.
- ***address*** is the address in ROM of ***rdp***.

## Description

`slot_long` returns the long contained in ROM which is pointed to by ***rdp***. ***address*** is the location in ROM of ***rdp***.

Pointers into ROM are calculated by adding the offset contained in the pointer to the ROM address. A resource data item of 32 bits can't be directly stored in the `rom_idoffset` structure (as both bytes and words can), so access to it must be granted indirectly. `slot_long` is used when the resource type of data stored in ROM is a 32- bit quantity. This routine as well as the `slot_byte`, the `slot_word`, and the `slot_structure` routines are used to access the four types of low-level ROM data types.

## Return values

| Value | Meaning |
|---|---|
| ***n*** | Success (the unsigned 32-bit value is returned). |
| None | All values are legal. |

## Example

```
u32 l;
char *address;
char *romp;
int j;
struct rom_idoffset rd[20];
/*
* Create a ROM base address.
*/
if((address = slot_address(i)) < 0) {
/* Error... */
}
/*
* Get the resource directory from ROM.
*/
if((romp = slot_directory(address, rd, 20)) == 0) {
/* Error... */
}
/*
* Find a resource of type long, and read the value into variable "l."
*/
for(j = 0; ((j < 20) && (rd[j].r_id != RD_EOLIST)); j++) {
        if( rd[j].r_id == RD_LONG) {
/*
* Update the pointer to point to the location in ROM of the
* long resource pointer.
*.
if((romp =
        slot_calc_pointer(romp, i*sizeof(struct rom_idoffset))) == 0) {
        /* Error */
```

```
        }
        l = slot_long(romp, rdp);
        break;
        }
}
```

## Name

`slot_resource`

## Synopsis

`char *slot_resource(`***address, kind, request, data, size***`)`

`char *`***address***`;`

`int` ***kind***`;`

`int` ***request***`;`

`char *`***data***`;`

`int` ***size***`;`

where

- ***address*** is base address for the slot ROM.
- ***kind*** is the resource list to be searched to find the ***request*** resource.
- ***request*** is the resource required by the calling process.
- ***data*** is a pointer to a buffer to hold the resource list.
- ***size*** is the number of `rom_idoffset` structures that can be stored in the user buffer.

## Description

`slot_resource` reads and returns up to ***size*** bytes of the structure associated with the resource of type ***request***. The requested resource must be located in the resource list of type ***kind***. The ***address*** parameter specifies the ROM base address to be read.

A resource list is a sublist of the ROM resource directory. Resources are substructures of resource lists. `slot_resource` reads the resource list of type ***kind*** into local storage using the library function `slot_resource_list`. The resource list is searched for the requested resource request and, if found, the associated resource structure is read and its contents returned into the user buffer. To read the ROM data, `slot_calc_pointer` is called to create a pointer to the base of the resource structure to be read, and then `slot_structure` is called to transfer the list from ROM to the user buffer.

## Return values

| Value | Meaning |
|---|---|
| *pointer* | Success (a pointer to the resource structure in ROM is returned). |
| 0 | Failure |

## Example

```
int slot = 9;
struct PRAM *Pp;
unsigned address;
/*
* Get the ROM base address
*/
if((address = slot_address(slot)) == 0) {
        /* Error */
}
/*
* Get the board resource list item of type RBL_PRAM from the
* resource list of type RD_BOARD in the resource directory.
*/
if(slot_resource(address, RD_BOARD, RBL_PRAM,
        Pp, sizeof(struct PRAM)) == 0) {
        /* Error */
}
```

## Name

`slot_resource_list`

## Synopsis

`char *slot_resource_list(`***address, kind, data, size***`)`

`char *`***address***;

`int` ***kind***;

`char *`***data***;

`int` ***size***;

where

- ***address*** is base address for the slot ROM.
- ***kind*** is the type of resource requested.
- ***data*** is a pointer to a buffer that holds the resource list.
- ***size*** is the number of `rom_idoffset` structures that can be stored in the user buffer.

## Description

`slot_resource_list` reads and returns up to ***size*** entries of the resource list associated with the resource of type ***kind***. The ***address*** parameter specifies the base address of the ROM to be read.

A resource list is a sublist of the ROM resource directory. `slot_resource_list` reads the resource directory into local storage using the library function `slot_directory`. The directory is searched for the requested resource ***kind*** and, if found, the associated resource list is read and its contents are returned in the user's buffer. To read the ROM data, `slot_calc_pointer` is called to create a pointer to the base of the resource list to be read, and then `slot_rom_data` is called to transfer the list from ROM to the user buffer.

## Return values

| Value | Meaning |
|---|---|
| *pointer* | Success (a pointer to the resource list in ROM is returned when the search and read of the resource list associated with the resource type ***kind*** is successful). |
| 0 | Failure |

## Example

```
int slot = 9;

unsigned address;

struct rom_idoffset rl[LISTLEN];

char *romp;

/*

* Create a ROM base address.

*/

if((address = slot_address(slot)) < 0) {

/* Error... */

}

/*

* Get the resource list for the resource of type RD_ETHER.

*

*/

if((romp = slot_resource_list(address, RD_ETHER, rl, LISTLEN)) == 0) {

/* Error */

}
```

## Name

`slot_structure`

## Synopsis

`int slot_structure(`***address, rdp, data, size***`)`

`struct rom_idoffset` ***rdp***;

`unsigned` ***address***;

`char` ****data***;

`int` ***size***;

where

- ***rdp*** is a structure defining a 32-bit quantity in ROM that contains a pointer to the structure to be read.
- ***address*** is the address in ROM of ***rdp***.
- ***data*** is a pointer to the user buffer to be filled with ROM data.
- ***size*** is the size (in bytes) of the user buffer.

## Description

`slot_structure` copies ***size*** bytes found in ROM at ***address*** plus the offset contained in the `rom_idoffset` structure ***rdp*** into the buffer pointed to by ***data***.

Pointers into ROM are calculated by adding the offset contained in the pointer to the ROM address. `slot_structure` is used when the resource type of data stored in ROM is a structure or string of an unknown size. This routine, the `slot_byte`, the `slot_word`, and the `slot_long` routines are used to access the four types of low-level ROM data types.

## Return values

| Value | Meaning |
|---|---|
| *count* | Success (the number of bytes of structure data is returned). |

–1 Failure

## Example

```
char data[100];
char *address;
char *romp;
int j;
struct rom_idoffset rd[20];
/*
* Create a ROM base address.
*/
if((address = slot_address(i)) < 0) {
/* Error... */
}
/*
* Get the resource directory from ROM.
*/
if((romp = slot_directory(address, rd, 20)) == 0) {
/* Error... */
}
/*
* Find a resource of some user type, and read the structure.
*/
for(j = 0; ((j < 20) && (rd[j].r_id != RD_EOLIST)); j++) {
if( rd[j].r_id == RD_USER) {
/*
* Update the pointer to point to the location in ROM of the
* resource pointer.
*/
if((romp =
```

```
slot_calc_pointer(romp, i*sizeof(struct rom_idoffset))) == 0) {
/* Error */
}
l = slot_structure(romp, rdp, data, sizeof(data));
break;
}
}
```

## Name

`slot_word`

## Synopsis

```
u16 slot_word(rwp)
        struct rsrc_word rwp;
```

where *wp* is a structure that defines a 32-bit quantity in ROM.

## Description

`slot_word` returns the 16-bit word contained in ROM that is part of the `rsrc_word` structure passed to the routine. The structure `rsrc_word` is defined in `slots.h`. The word returned contains the least significant 16 bits of a 32-bit quantity defined by the `rsrc_word` structure.You use `slot_word` when the resource type of data stored in ROM is a unsigned 16-bit quantity. This routine, as well as the `slot_long`, the `slot_byte`, and the `slot_structure` routines are used to access the four types of low-level ROM data types.

## Return values

| Value | Meaning |
|---|---|
| *data* | Success (the 16-bit word of data stored in *rup* is returned). |
| None | All values are legal. |

## Example

```
struct rsrc_word rwp;

u16 w;

char *address;

int j;
```

```
struct rom_idoffset rd[20];
/*
 * Create a ROM base address.
 */
if((address = slot_address(i)) < 0) {
/* Error... */
}
/*
 * Get the resource directory from ROM.
 */
if(slot_directory(address, rd, 20) < 0) {
/* Error... */
}
/*
 * Find a resource of type WORD, and read the value into variable c.
 */
for(j = 0; ((j < 20) && (rd[j].r_id != RD_EOLIST)); j++) {
if( rd[j].r_id == RD_WORD) {
rwp = (struct rsrc_word)rd[j];
w = slot_word(rbp);
break;
}
}
```

## Name

`slot_seg_violation`

## Synopsis

```
slot_seg_violation()
```

## Description

`slot_seg_violation` protects the slot library functions from illegal ROM accesses. It is passed to the `slot_catch` routine to catch user segmentation violations and to allow error recovery.

`slot_env` is a initialized environment structure used with a UNIX `longjmp` call.

## Return values

There is no return; the program makes a `longjmp` call to a prearranged error-handling routine.

## Example

```
main() {
char *romp;
int slot = 9;
/*
* Create a ROM address.
*/
if((romp = slot_address(i)) < 0) {
/* Error... */
}
```

```
/*
 * prepare for ROM access timeouts. First catch the segmentation
 * violation signal.
 */
slot_catch(SIGSEGV, slot_seg_violation);
/*
 * Initialize slot_env and test to see how we got here. If true,
 * an error occured. If false, then the initialization is
 * complete.
 */
if(setjmp(slot_env)) {
/*
 * Error, caught a segmentation violation. Reset the signal
 * handler for segmentation violations then exit in error.
 */
slot_ignore(SIGSEGV);
exit(1);
}
/*
* Try reading the ROM pointed to by "romp." Errors will cause a
* branch back to the setjmp.
*/
c = *romp & 0xf;
/*
* If the code gets to here, the ROM is readable. Reset the error
* handler, and exit with good status.
*/
slot_ignore(SIGSEGV);
exit(0);
}
```

## Name

`slot_catch`

## Synopsis

`slot_catch(`***kind, routine***`)`

`int` ***kind;***

`int *`***routine***`();`

where

- ***kind*** is the signal type.
- ***routine*** is the error recovery handling routine that is called.

## Description

`slot_catch` uses the `signal` system call to initalize nonstandard signal handling for a signal of type ***kind***. The result of the `signal` call is that interrupts of type ***kind*** cause the error recovery handling ***routine*** to be called.

`slot_env` is a preinitialized environment structure used with a `longjmp` call.

## Return values

None.

## Example

```
main() {
char *romp;
int slot = 9;
/*
* Create a ROM address.
```

```
*/
if((romp = slot_address(i)) < 0) {
/* Error... */
}
/*
* prepare for ROM access timeouts. First catch the segmentation
* violation signal.
*/
slot_catch(SIGSEGV, slot_seg_violation);
/*
* Initialize slot_env and test to see how we got here. If true,
* an error occured. If false,then the initialization is complete.
*/
if(setjmp(slot_env)) {
/*
* Error, caught a segmentation violation. Reset the signal
* handler for segmentation violations then exit in error.
*/
slot_ignore(SIGSEGV);
exit(1);
}
/*
 * Try reading the ROM pointed to by "romp." Errors will cause a
 * branch back to the setjmp.
 */
c = *romp & 0xf;
/*
* If the code gets to here, the ROM is readable. Reset the error
* handler, and exit with good status.
*/
```

```
slot_ignore(SIGSEGV);
exit(0);
}
```

## Name

`slot_ignore`

## Synopsis

`slot_ignore(`***kind***`)`

`int` ***kind***;

where ***kind*** is the signal to restore to default handling.

## Description

`slot_ignore` uses the `signal` routine to restore default signal handling for signals of type ***kind***.

## Return values

None.

## Example

```
main() {
char *romp;
int slot = 9;
/*
* Create a ROM address.
*/
if((romp = slot_address(i)) < 0) {
/* Error... */
}
/*
```

```
 * prepare for ROM access timeouts. First catch the segmentation
 * violation signal.
 */
slot_catch(SIGSEGV, slot_seg_violation);
/*
* Initialize slot_env and test to see how we got here. If true,
* an error occured. If false, then the initialization is
* complete.
*/
if(setjmp(slot_env)) {
/*
 * Error, caught a segmentation violation. Reset the signal
 * handler for segmentation violations then exit in error.
 */
slot_ignore(SIGSEGV);
exit(1);
}
/*
 * Try reading the ROM pointed to by "romp." Errors will cause a
 * branch back to the setjmp.
 */
c = *romp & 0xf;
/*
* If the code gets to here, the ROM is readable. Reset the error
* handler, and exit with good status.
*/
slot_ignore(SIGSEGV);
exit(0);
}
```

## Name

`slot_address`

## Synopsis

`char *slot_address(`*slot*`)`

`int` *slot*;

where *slot* is either a slot number, a physical ROM base address, or a virtual ROM base address.

## Description

`slot_address` checks *slot* for validity and type, and returns a valid ROM base address for *slot*.. Physical and virtual addresses are returned directly if the calling routine is valid. A slot input parameter is converted to a physical address, and the ROM at that address is made available to the user program by using the `phys` system call (which makes the ROM available on a A/UX page boundary). A page in the system is 4 megabytes, so `phys` is called to map the slot ROM to a virtual address of 4 megabytes.

## Return values

None.

## Example

```
main() {
char *romp;
int slot = 9;
/*
* Create a ROM address.
*/
```

```
if((romp = slot_address(i)) < 0) {
/* Error... */
}
/*
* prepare for ROM access timeouts. First catch the segmentation
* violation signal.
*/
slot_catch(SIGSEGV, slot_seg_violation);
/*
 * Initialize slot_env and test to see how we got here. If true,
 * an error occured. If false, then the initialization is
 * complete.
 */
if(setjmp(slot_env)) {
/*
* Error, caught a segmentation violation. Reset the signal
* handler for segmentation violations then exit in error.
*/
slot_ignore(SIGSEGV);
exit(1);
}
/*
 * Try reading the ROM pointed to by "romp." Errors will cause a
 * branch back to the setjmp.
 */
c = *romp & 0xf;
/*
 * If the code gets to here, the ROM is readable. Reset the error
 * handler, and exit with good status.
 */
```

```
slot_ignore(SIGSEGV);
exit(0);
}
```

## Name

`slot_byte_lane`

## Synopsis

`char *slot_byte_lane(`*address, byte lane*)

`char *`*address*;

`char *`*byte-lane*;

where

- *address* is either a physical ROM base address or a virtual ROM base address.
- *byte-lane* is a pointer to a location that stores byte lane information from ROM.

## Description

`slot_byte_lane` searches ROM starting at *address* for the byte lane byte that should be located in the last byte in addressable ROM (such as. 0xFssFFFFF, or *address* plus 0xFFFF). The search continues backwards for up to 4 bytes to allow for the possible board addressing conventions. When a valid byte lane byte is located, that information is stored in the location pointed to by the parameter *byte lane*, and the address in ROM of the byte lane byte is returned. If no valid byte lane byte is found, an error is returned.

A byte lane byte contains the valid addresses using NuBus addressing conventions for ROM data. The `slot_byte_lane` routine reverses the byte lane information (4 bits) before returning the data to the caller. because the Motorola 68000 family uses an addressing convention that is reversed from the NuBus standard. `slot_calc_pointer` uses the byte lane information to create valid ROM addresses. The format of the byte lane byte may be found in *Developing Cards and Drivers for Macintosh II and Macintosh SE* .

## Return values

| Value | Meaning |
|---|---|
| *address* | Success (the address of the byte lane information in ROM is returned). |

0 Failure

## Example

```
char *curr, *base;
char bl;
/*
 * Get the byte lane byte using the slot_byte lane routine.
 * If no byte lane is available, you can't calculate the
 * pointer.
 */
if((base = slot_byte lane(((unsigned)curr & 0xFFFF0000), &bl)) == 0) {
 /* Error */
}
```

## Name

`slot_calc_pointer`

## Synopsis

`char *slot_calc_pointer(`*current*, *offset*)

`char *`*current*;

`int` *offset*;

where

- *current* is the current ROM address.
- *offset* is the offset, in bytes.

## Description

`slot_calc_pointer` calculates the valid ROM address that is *offset* bytes from *current*. The address is a byte address in slot ROM for a board in one of the Macintosh II NuBus slots. `slot_calc_pointer` calls `slot_byte_lane` to get the byte lane information for the ROM pointed to by *current*. Using the byte lane information, and adding *offset* to *current*, a new ROM address is created and returned to the user.

The *offset* is a count of bytes "skipped" to get to the new ROM address. The count may be positive or negative. The bytes to be skipped are not necessarily in continuous memory (that is, a simple add of address and offset will skip *offset* "addresses") but if only one out of every four bytes is active, then only a quarter of the offset is fullfilled. In the active byte lanes, the offset must be multiplied by four to skip the full number of *offset* bytes. The simple addition of address and *offset* works when all byte lanes are active (not the common case). There is also a small calculation required to land on an active ROM address based on the value of *current*, because *offset* may not be an even multiple of four.

## Return values

| Value | Meaning |
|---|---|
| *address* | Success (the new ROM address is returned). |

0 Failure

## Example

```
char *romp;
int offset = 10;
int slot = 9;
/*
 * Create a ROM address.
 */
if((romp = slot_address(i)) < 0) {
/* Error... */
}
if((romp = slot_calc_pointer(romp, offset) == 0) {
/* Error */
}
/*
 * ROMP is now offset bytes from the base of ROM.
 */
```

## Name

`slot_rom_data`

## Synopsis

`char *slot_rom_data (`***address, width, data***`)`

`char` ****address;***

`int` ***width;***

`char` ****data;***

where

- ***address*** is a ROM address.
- ***width*** is a positive or negative count of bytes to read.
- ***data*** is a pointer to the user buffer to be filled.

## Description

`slot_rom_data` fills the buffer pointed to by ***data*** with ***width*** bytes of data from ROM starting from ***address***.

`slot_rom_data` reads the byte lane information using the `slot_byte_lane` routine. Reading only valid ROM addresses, `slot_rom_data` reads the data from ROM and stores it into the user buffer referenced by ***data***. The direction of the read is determined by the value of ***width***. A negative value causes the bytes to be read in reverse order from address to (***address–width***.), and a positive width causes a read of ROM data from address to (***address + width***). The positive count of bytes read from ROM is returned to the user.

`slot_env` is a preinitialized environment structure to use with a `longjmp` call.

## Return values

| Value | Meaning |
|---|---|
| *count* | Success (a non-negative count of bytes read is returned). |

-1 Failure

## Example

```
char *romp;
struct format_header fhp;
int slot = 9;
/*
 * Create a ROM address.
 */
if((romp = slot_address(i)) < 0) {
     /* Error... */
}
if((romp = slot_byte lane(base, &byte lane)) == 0) {
       /* Error */
}
/*
 * Read the ROM format header, again errors cause a zero return.
 */
if((count = slot_rom_data(romp,-sizeof(struct format_header),fhp)) < 0) {
       /* Error */
}
```

## Name

`slot_check_crc`

## Synopsis

`char *slot_check_crc(`*top, fhp, byte lane*`)`

`char *`*top*;

`struct format_header *`*fhp*;

`char` *byte-lane*;

where

- *top* is the address of a byte lane byte in slot ROM.
- *fhp* is a pointer to the format header structure from slot ROM.
- *byte-lane* is the 4 bits of byte lane information from slot ROM.

## Description

`slot_check_crc` computes and verifies the ROM checksum for the slot ROM ending at address *top*.

`slot_check_crc` is called from `slot_header`, which reads and verifies both the the lowest level ROM structure and the ROM contents. The format header and byte lane information for the ROM to be checked are read by `slot_header` and then passed to the checksum routine. The format header contains the ROM length and the ROM checksum to be verified. The byte lane byte contains the addressing information used to read the ROM.

`slot_env` is a preinitialized environment structure used with a `longjmp` call.

## Return values

| Value | Meaning |
|---|---|
| 0 | Success (checksum is valid). |
| 1 | Failure |

## Example

```
struct format_header fhp;
char *romp;
char byte lane;
int count;
/*
 * Get the byte lane byte, and a pointer to
 * the location of the byte lane byte in ROM.
 */
romp = slot_byte lane(base, &byte lane);
if(romp == (char *) 0) {
        /* Error */
}
/*
 * Read the format header, again errors cause a zero return.
 */
if((count = slot_rom_data(romp,-sizeof(struct format_header),&fhp)) < 0){
        /* Error */
}
/*
 * Check that the format header contains valid information.
 */
if((fhp->f_testpattern != F_TESTPATTERN) || (fhp->f_rev > F_REV) ||
(fhp->f_format != F_APPLE) || (fhp->f_reserved != 0) ||
((fhp->f_diroffset & 0x00FFFFFF) == 0) ||
((fhp->f_diroffset & 0xFF000000) != 0))
return(0);
/*
 * check the checksum.
 */
```

```
if(slot_check_crc(romp, &fhp, byte lane) != 0) {
        /* Error */
}
```

## Name

`slot_header`

## Synopsis

`char *slot_header(`*address, fhp*`)`

`char *`*address*;

`struct format_header *`*fhp*;

where

- *address* is either a physical ROM base address or a virtual ROM base address.
- *fhp* is a pointer to a format header structure.

## Description

`slot_header` reads and verifies the ROM header and ROM checksum, storing the information, if valid, in the buffer pointed to by *fhp*. *address* is the base address of the slot ROM.

The ROM header describes the slot ROM. It contains a ROM checksum, the ROM length, a pointer the resource directory, and information words.

## Return values

| Value | Meaning |
|---|---|
| *pointer* | Success (a pointer to the format header structure in ROM is returned). |
| 0 | Failure |

## Example

```
struct format_header fh;

char *romp;

int slot = 9;
```

```
/*
* Create a ROM address.
*/
if((romp = slot_address(i)) < 0) {
/* Error... */
}
/*
* Get the ROM header into "fh."
*/
if((romp = slot_header(address, &fh)) == 0) {
/* Error */
}
```

# Appendix D

# Memory Maps

This appendix contains memory maps for the Macintosh II and A/UX. Included are the memory maps for the physical address space, user address space, and kernel address space.

**Physical address space**

The physical address space for the Macintosh II is shown in Figure D-1.

**Figure D-1**
Physical address space

## User address space

In A/UX, the user address space contains 512 megabytes. This means that the kernel keeps the entire user process in the kernel address space. The user address space is shown in Figure D-2.

**Figure D-2**
User address space

## Kernel address space

For A/UX device driver writers, the kernel address space is shown in Figure D-3.

**Figure D-3**
Kernel address space

# Appendix E

# Vnode Kernel Driver Modifications

Although the A/UX kernel is based on AT&T's System V Release 2, the A/UX device driver interface is closer to that used in U.C. Berkeley's 4.2 BSD. This section notes some changes you should be aware of if you are familiar with the AT&T driver interface.

A/UX supports the vnode kernel. Because of this, device driver interfaces differ from those in other UNIX systems. If you are familiar with UNIX device drivers in other systems or are porting a driver from another system to A/UX, be aware of the following A/UX driver changes.

- Driver return values: a driver's open, read, write, and functions return error values differently. Berkeley-style drivers require these routines to return either 0 for success or an error number (defined in `<sys/errno.h>`) for failure. Normally, AT&T System V drivers set the global value `u.u_error` to indicate failure and nothing to indicate success.
- Device read and write interfaces: the vnode kernel passes I/O parameters in a `uio` structure. Other drivers use `rdwr` to initialize the values `u.u_base`, `u.uoffset`, and `u.u_segflg` before calling the device specific read or write function. For details about the `uio` data structure, see "Device Read and Write Interfaces" in Chapter 2.
- I/O control interface: I/O control in A/UX encodes information about whether the I/O request will copy data in or out of kernel memory (or both) and the amount of data that will be copied (if any). Thus, an `ioctl`(2) system call can do `copyin` or `copyout` itself, rather than passing to the device I/O control interface a pointer to a buffer in kernel memory. For details about this interface, see "I/O Control Interface" in Chapter 4.

- Disk performance monitoring: to monitor disk drives, several global variables have been provided to keep track of disk performance. See "Monitoring Disk Performance" in Chapter 3 for details.

# Appendix F

# V.2 Streams Drivers

This appendix lists the differences between AT&T's System V Release 3 and System V Release 2.1 Streams drivers supported by A/UX. Specifically, System V Release 2.1 doesn't support the following System V Release 3 features:

- input/output polling
- asynchronous input/output
- multiplexed streams
- `putmsg` and `getmsg` routines
- services interfaces and messages
- `bufcall`, `enableok`, `datamsg`, `insq`, `noenable`, `pullupmsg`, `rmvq`, and `testb` utility routines
- `NSTREVENT`, `MAXSEPGCNT`, `NSTRPUSH`, `STRMSGSZ`, `STRCTLSZ`, `STRLOFRAC`, and `STRMEDFRAC` system parameters

In addition, the following enhancements have been added to the A/UX Streams implementation.

- `ttx` data structure: A/UX uses the `ttx` data structure to hold terminal information.
- `ttx` library: to make it easier to write terminal drivers, A/UX provides the `ttx` library, a collection of Streams support routines.
- `line`: the streams line discipline for terminals.
- `select`: A/UX provides support for the `select`(2) BSD system call.
- FIO...ioctl(s) are supported. See `streams`(7) in the *A/UX System Administrator's Reference* for details.

# Appendix G

# SCSI Device Driver

This appendix contains a SCSI driver source listing for a hard disk. The driver consists of two main parts: the generic disk driver and the SCSI manager. The SCSI driver files included in this appendix are illustrated in Figure G-1.

See Chapter 10 for detailed descriptions of specific generic disk driver and SCSI manager routines, including parameters and calling sequences.

**Figure G-1**
The SCSI Driver

## Generic disk driver files

The generic disk driver consists of the following files:

- hd.c—the high-level driver interface to the generic disk driver
- gdisk.c—the generic disk driver routines
- gdisksubr.c—the generic disk driver subroutines

The hd.c file contains the driver interface to the kernel. These are the routines called through the bdevsw table. The rouines in hd.c, in turn call the gdisk.c routines, which interface to the generic disk driver functions based on a Finite State Machine (located in the file gdisksubr.c). The machine coordinates general I/O tasks as generic state sequences. This results in function calls to routines in the file sdisk.c, which contains low-level SCSI routines.

Note: If you're writing a NuBus disk driver, your driver can interface to the generic disk driver files in the same way as a SCSI disk driver. You must, however, write different device-specific bdevsw and low-level routines to replace hd.c, sdisk.c, and scsi.c.

## SCSI manager files

The SCSI manager is contained in these two files:

- sdisk.c
- scsi.c

The sdisk.c and scsi.c files form the lower level of the driver-specific portion of the driver. sdisk.c contains the low-level device-specific interface between the generic disk driver and the SCSI manager. It contains the routines sdread, sdwrite, sddriveinit, sdbadblock, sdformat, sdrecover, and sdshutdown, which are described in "Low-level Device Routines" in Chapter 10.The scsi.c code contains the low-level routines that implement the SCSI manager functions (as another Finite State Machine), as well as routines that drive the NCR5380 chip.

## Other files

In addition to the generic disk driver and SCSI manager files just listed, you may find the following files useful while writing your driver:

- `ncr5380.h` NCR register definitions
- `via6522.h` NCR interrupt decoding information
  `via.c`
- `gdisk.h` Disk task blocks definitions

## The hd.c source code

The following pages list the source code for hd.c.

## The gdisk.c source code

The following pages list the source code for gdisk.c.

## The gdisksubr.c source code

The following pages list the source code for gdisksubr.c.

## The sdisk.c source code

The following pages list the source code for sdisk.c.

## The scsi.c source code

The following pages list the source code for scsi.c.

# Glossary

**autoconfiguration:** A technique for adding, deleting, or replacing a device driver in the A/UX kernel.

**backenable:** A method of scheduling a Streams queue for service by preventing new messages from being scheduled after a high water mark is reached and allowing new messages to be scheduled only after the number of messages on a queue have dropped below a low water mark.

**block devices:** Devices that access data blocks, which permits them to contain mounted file systems. Reading and writing to block devices are handled through a cache of buffers that minimize physical access to the device.

**character devices:** Devices that generally perform I/O asynchronously for a variable number of bytes.

**cblock:** A data structure used to buffer terminal data. Cblocks are linked together to form a clist queue. A cblock contains an array to hold data, pointers to the first and last characters in the array, and a pointer to the next cblock on the queue.

**clist:** The basic terminal buffering structure. A clist is the head of a linked list queue of cblocks.

**device class:** A class of device that share data access characteristics. In A/UX possible devices classes are block, character, or network.

**device driver:** A portion of the kernel that handles I/O operations to and from a physical device in the system.

**device interface:** In A/UX, the device driver's interface to the device itself. Possible interface types include NuBus, SCSI, and ADB.

**device type:** A specific kind of A/UX device. With a device class, there can be several different device types.

**driver interface:** The A/UX kernel's interface to a device driver. Possible interface types include block, character, terminal, streams, and network.

**high-water mark:** For terminal I/O, the maximum number of characters that can be in the raw queue before input is temporarily suspended. For Streams I/O, it is used along with the low water mark to schedule a queue.

**line discipline:** A data structure containing pointers to a terminal's open, close, read, write, ioctl, input interrupt, and output interrupt routines.

**loadfile:** A file needed to run autoconfiguration in a driver development environment. It contains slot ROM information normally found in the system's slot ROMs.

**low-water mark:** For terminal I/O, the number of characters in the raw queue must drop below this level before additional characters can be added to the queue. For Streams I/O, it is used along with the high water mark to schedule a queue.

**makefile:** A file containing user-specified commands that are processed according to built-in rules contained in the `make`(1) utility.

**master script file:** A file that contains information used during autoconfiguration.

**module:** In Streams, a pair of queues that process data traveling between the stream head and the Streams driver.

**network devices:** Devices that handle data communication between machines.

**process:** In A/UX, an instance of a program in execution.

**queue:** In Streams, a data structure that is associated with a statically compiled module. Queues are always found in pairs—one for upstream processing and one for downstream processing.

**raw interface:** A character interface that handles reading and writing to a device directly, without buffering data. Block devices use both a raw and a buffered interface.

**request block data structure:** A data structure that specifies the elements of a SCSI command. Instead of sending a SCSI command directly to the controller, the request block structure is filled and passed to the SCSI manager for processing.

**stream:** A full duplex processing and data transfer path between a driver in kernel space and a process in user space.

**Streams:** A collection of system calls, kernel resources, and kernel utility routines that can create, use, and dismantle a stream. Streams provides a convenient mechanism for writing an A/UX terminal driver.

**Streams driver:** A part of the stream end that performs device handling and also transforms data and information that passes between the external interface and a stream.

**Streams module line:** A line discipline used in Streams terminal drivers to perform such functions as echoing characters, providing erase and kill processing, flow control, ioctl processing, and character editing.

**Stream end:** The part of a stream closest to the external device interface. The stream end contains the Streams driver.

**Stream head:** The part of a stream that provides the interface between the stream and the user process.

**Streams messages:** The form in which blocks of data are linked together and passed through a stream. Each message block consists of data structures and a buffer block.

**transaction:** The most basic function that a driver requests of the ADB. It consists of a request for the ADB, an ADB operation, and a reply from the ADB after the transaction has completed.

**ttx structure:** A data structure used in A/UX Streams terminal drivers that contains information needed for terminal I/O.

**tty structure:** A data structure containing information needed to perform terminal I/O. This includes pointers to raw, canonical, and output queues; and a pointer to a device driver command processing routine.

**u-dot:** A data structure containing information and pointers unique to a process. The u-dot is also called the user structure.

**uio structure:** A data structure that describes a data transfer. The uio structure, an argument to several routines, contains read and write parameters.

# Bibliography

The following reading list contains additional documents that you might find useful when writing an AU/X device driver. Documents are listed according to the topic they cover.

## General Information

Bach, Maurice J., *The Design of the UNIX Operating System*. Englewood Cliffs: Prentice-Hall, 1986. This book describes the major elements of the UNIX operating system from a programmer's perspective. Chapter 10 describes the device driver interfaces, with special attention paid to terminal and disk drivers.

Motorola Corporation. *MC68020 32-Bit Microprocessor User's Manual*, 2nd. ed., Englewood Cliffs: Prentice-Hall, 1985. This book describes the operating features of the MC68020 microprocessor.

Motorola Corporation. *MC68881 Floating-Point Coprocessor User's Manual*. Englewood Cliffs: Prentice-Hall, 1985. This book describes the operating features of the MC68881 floating-point coprocessor.

Motorola Corporation. *MC678851 Paged Memory Management Unit User's Manual*. Englewood Cliffs: Prentice-Hall, 1986. This book describes the operating features of the MC68881 paged memory management unit.

## SCSI device drivers

*X3T9.2 Small Computer System Interface*, Revision 17B, 1985. This paper describes the Small Computer System Interface (SCSI).

ANSI Standards Committee. *Common Command Set of the Small Computer System Interface*, Revision 4.A. This paper describes the common command set of the SCSI interface.

NCR Micro Electronics Division, *NCR 5380 SCSI Interface Chip Design Manual*. Colorado Springs: NCR Microelectronics Division, 1985. This book describes the design features of the NCR 5380 SCSI interface chip.

## Streams device drivers

AT&T Corporation. *STREAMS Programmer's Guide*. 1986. This book contains comprehensive information about Sreams utilities and functions. It should be read before attempting to write a Streams device driver.

AT&T Corporation. *STREAMS Primer*. 1986. This book defines and gives a brief overview of Streams.

## Network device drivers

Leffler, Samuel J., Robert S.Fabry, and William N. Joy. "Networking Implementation Notes." University of California, Berkeley. This paper describes the internal structure of 4.2BSD-style networking facilities.

## Other A/UX documents

Apple Computer, Inc. *A/UX Programmer's Reference*: 1987. This manual is a quick reference guide to all library routines and related information required by programmers, as well as miscellaneous facilities.

Apple Computer, Inc. *A/UX System Administrator's Reference*: 1987. This manual is a quick reference guide to privileged commands and utitlity programs that would be used by the system administrator.

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word. Proof pages were created on the Apple LaserWriter® Plus. Final pages were created on the Varityper® VT600™. POSTSCRIPT®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated. Some of the illustrations were created using Adobe Illustrator™.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier, a fixed-width font.

**2/21/88**

**note: add Illustrator™ to the list of credits on the copyright page.**