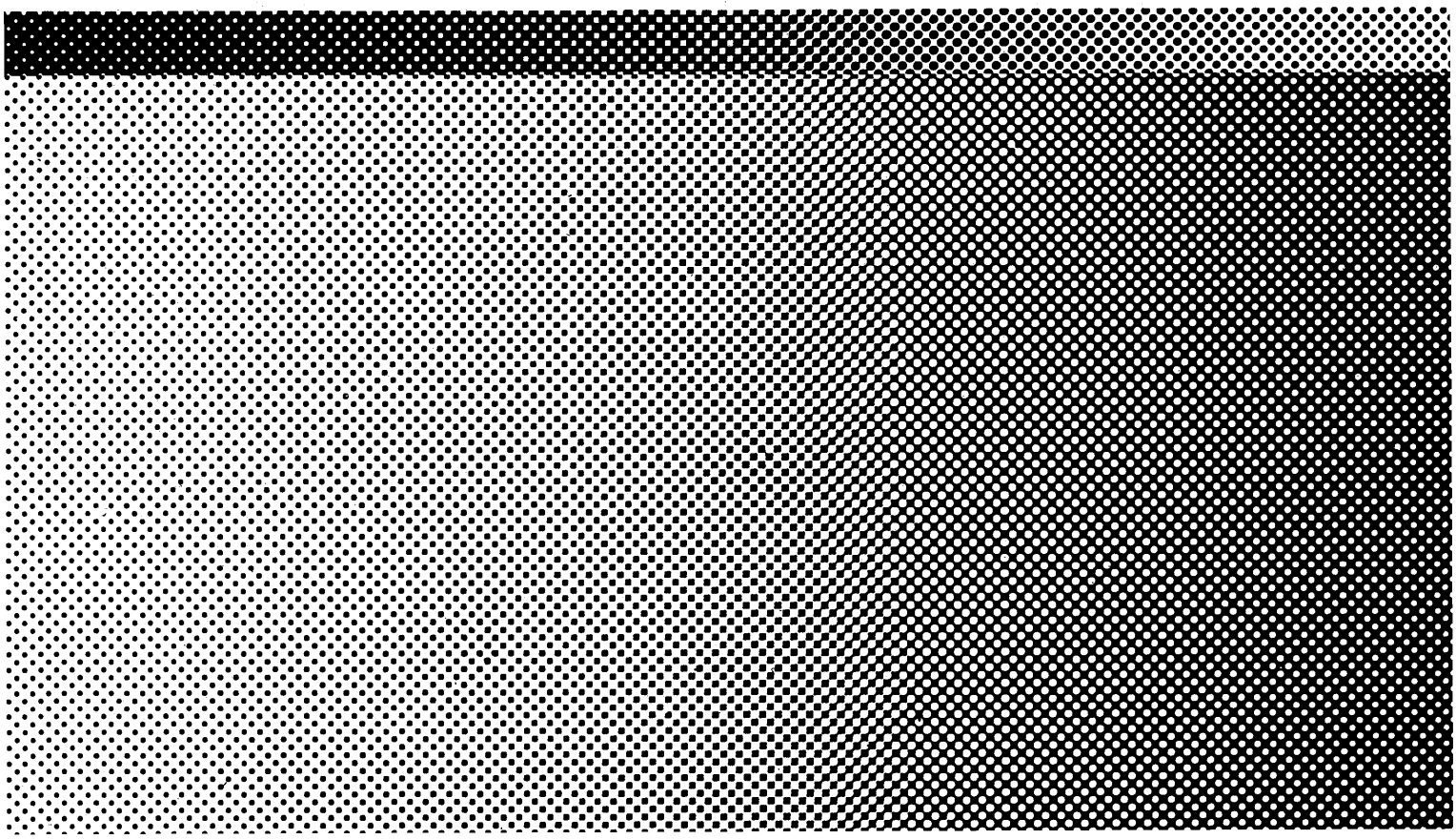
UNIX® System V, Release 3
Block and Character Interface (BCI) Driver Development Guide

307-191
Issue 1
Preliminary

UNIX® System V, Release 3
Block and Character Interface (BCI) Driver Development Guide



Notice

Information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

For ordering information on this document or related learning support materials, see "Related Learning Support Materials," in "About This Document."
The ordering number for this document is 307-191

Contents

Installation

Testing and Debugging the Driver

Performance Considerations

Porting Drivers

Packaging the Driver

Appendix A: Equipped Device Table (EDT)

Appendix B: Writing 3B2 Computer Diagnostics Files

List of Figures

List of Tables

Chapter 1: About This Document

Contents

About This Document

The *AT&T Block and Character Interface (BCI) Driver Development Guide* (shortened hereafter to *BCI Driver Development Guide*) provides information needed to write, install, and debug drivers in the UNIX® System V environment. It supplements the *AT&T Block and Character Interface (BCI) Driver Reference Manual* (shortened hereafter to *BCI Driver Reference Manual*) with general information and guidelines on writing, installing, and debugging drivers. It also includes background information on such topics as how drivers are configured into the operating system at boot time, how the operating system accesses driver entry point routines, and the different I/O transfer schemes (with or without kernel buffering). For more information about this document, see the "How to Use This Document" section in this chapter.

Driver Development Series

The *BCI Driver Development Guide* is part of the *AT&T Driver Development Series*. The *Block/Character Interface (BCI) Driver Reference Manual* is a companion manual to this book. Other documents in this series include the *AT&T Portable Driver Interface (PDI) Reference Manual* and the *AT&T SCSI Driver Interface (SDI) Reference Manual*, which are listed in the "Related Documents" section at the end of this chapter.

Systems Supported

This document supports driver development among many different AT&T computers. Although most of the information presented in this book is applicable to any UNIX System V computer, the manual contains examples and information specifically for the following computers and releases:

- *WE*® 321SB Single-Board-Computer (SBC), UNIX System V/VME Release 3.1
- AT&T 3B2/300 Computer, UNIX System V Release 3.1
- AT&T 3B2/400 Computer, UNIX System V Release 3.1
- AT&T 3B2/500 Computer, UNIX System V Release 3.1
- AT&T 3B2/600 Computer, UNIX System V Release 3.1
- AT&T 3B15 Computer, UNIX System V Release 3.1.1
- AT&T 3B4000 Computer, UNIX System V Release 3.1.1

Note the following about textual references to various systems:

- The term *3B2* computer is used for information that is the same for all models of the 3B2 computer. The model number is specified only when information is not the same for all models.
- The 3B15 computer and 3B4000 Master Processor (MP) share the same kernel, so most driver information that pertains to one pertains to both. When the information is applicable to only one or the other system, it is so stated.
- The term *adjuncts* applies to the Adjunct Communications Processor (ACP), Adjunct Data Processor (ADP), and 3B4000 Enhanced Adjunct Data Processor (EADP). Information that is applicable to only certain adjuncts is so marked.

Purpose

The *BCI Driver Development Guide* provides the information needed to write, install, and debug device drivers in the UNIX System V environment.

Intended Audience

Both this book and the *BCI Driver Reference Manual* are written for advanced C programmers who write and maintain UNIX system drivers.

Prerequisite Skills and Knowledge

It is assumed that you are proficient with the advanced capabilities of the C programming language (including bit manipulation, structures, and pointers) and familiar with UNIX system internals. A number of documents and courses on these topics are available from AT&T. They are listed later in this chapter.

How to Use This Document

Figure 1-1 is a high-level roadmap to the topics covered in this book.

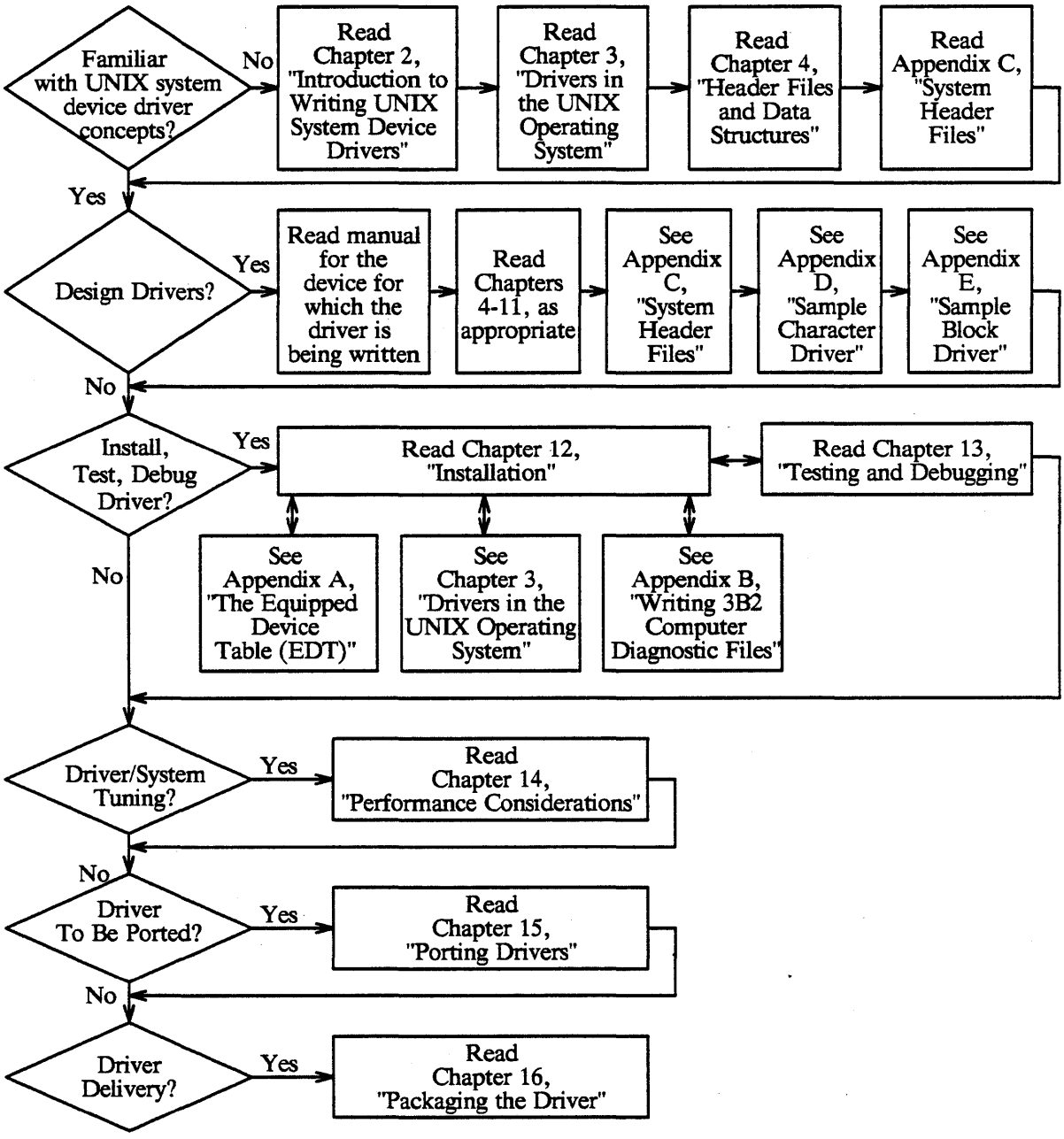


Figure 1–1 Roadmap to this Document

This rest of this chapter describes the conventions used in this document, related learning support materials and how to order them, and how to give us your comments about the *BCI Driver Development Guide*.

After this introductory chapter, this manual is organized as follows:

Chapter 2, *Introduction to Writing UNIX System Drivers*
: describes the process of writing a driver, including an outline of steps taken and general guidelines that driver writers should follow.

Chapter 3, *Drivers in the UNIX Operating System*
: discusses how master files are created and how drivers interface with the operating system.

Chapter 4, *Header Files and Data Structures*
: describes the use of system and driver-specific header files, and the relationship between data structures and drivers. Chapter 4 introduces some standard system header files delivered with the UNIX operating system that define error code, parameter, and data structure information for all drivers, and describes the standard system data structure fields frequently accessed by driver routines.

Chapter 5, *System and Driver Initialization*
: discusses the self-configuration and system initialization processes. System initialization initializes the kernel and drivers, creates process 0, executes the **init**(1M) process, and starts the system processes.

Chapter 6, *Input/Output Operations*
: provides general information on data transfer methods between the kernel and devices, and between user space and the kernel; detailed information on block data transfer methods, including information on character or physical I/O for a block device; detailed information on character data transfer methods, including information on buffered and unbuffered character I/O, and on allocating local driver memory; detailed information on creating a private buffering scheme; information on processor-specific memory management facilities; and information on scatter/gather I/O implementations.

Chapter 7, *Drivers in the TTY Subsystem*
: describes the components of the TTY subsystem. The TTY subsystem is a collection of functions and the driver **proc**(D2X) routine that are used to transfer information character-by-character between a CPU and a peripheral, such as a terminal or printer.

Chapter 8, *Input/Output Control (ioctl)*
: discusses the **ioctl** routine, which usually controls device hardware parameters and establishes the protocol used by the driver, and its relationship to the **ioctl**(2) system call.

Chapter 9, *Synchronizing Hardware and Software Events*
discusses how to use kernel functions, such as **sleep** and **wakeup**, that synchronize hardware and software events.

Chapter 10, *Interrupt Routines*
discusses servicing interrupts, preventing interrupts, interrupt vectors, and writing interrupt routines.

Chapter 11, *Error Reporting*
introduces interrupt handling and provides guidelines for writing interrupt handling routines.

Chapter 12, *Installation*
describes how to compile and install a driver and remove it from the system.

Chapter 13, *Testing and Debugging the Driver*
describes the general testing process and the debugging tools that are available for driver writers. It also discusses common driver bugs and gives suggestions for resolving them.

Chapter 14, *Performance Considerations*
discusses ways of checking and improving the performance of your driver as well as information on modifications that may be needed to maintain acceptable system performance when your driver is installed.

Chapter 15, *Porting Drivers*
summarizes the machine-specific features that must be considered when porting drivers among machines and provides instructions for writing a driver that ports easily between machines.

Chapter 16, *Packaging the Driver*
summarizes what to include in the software package that includes driver code.

Appendix A, *The Equipped Device Table (EDT)*
describes the Equipped Device Table (EDT). The EDT is a table in the private memory associated with the CPU that lists all hardware devices present on the system (except memory cards/boards).

Appendix B, *Writing 3B2 Computer Diagnostics Files*
explains how to write the files that test the integrity of a 3B2 computer feature card.

Appendix C, *System Header Files*
lists the system header files (from */usr/include/sys* directory and subdirectories) that can be used in driver code. It includes a number of header files for system data structures and structures associated with drivers that are bundled with the UNIX operating system.

Appendix D, *Sample Character Driver*
provides the code for a serial driver that interacts with a Dual Universal Asynchronous Receiver-Transmitter (DUART), such as that used by a terminal.

Appendix E, *Sample Block Driver*
provides the code for a disk controller driver (**doc_driver**) that runs on the SBC computer. This is an example of a hardware driver for a block-access device that also supports character access.

A *Glossary* and *Index* are also included at the end of this book.

Conventions Used in This Document

Table 1-1 lists the textual conventions used in this book. These conventions are also used in the *BCI Driver Reference Manual.*

Table 1–1 Textual Conventions Used In This Book

Item	Style	Example
C Bitwise Operators (| &)	CAPITALIZED	OR
C Commands	**Bold**	**typedef**
C **typedef** Declarations	**Bold**	**caddr_t**
Driver Routines	**Bold**	**strategy** routine
Error Values	CAPITALIZED	EINTR
File Names	*italics*	*/usr/include/sys/conf.h*
Flag Names	CAPITALIZED	B_WRITE
Kernel Macros	**Bold**	**minor**
Kernel Functions	**Bold**	**ttopen**
Kernel Function Arguments	*Italics*	*bp*
Keyboard Keys	Key	CTRL-d
Structure Members	**Bold**	**u_base**
Structure Names	`Constant Width`	`tty` structure
Symbolic Constants	CAPITALIZED	NULL
UNIX System C Commands	**Bold** (section reference)	**ioctl**(2)
UNIX System Shell Commands	**Bold**	**layers**(1)
User-Defined Variable	*Italics*	*prefix***close**

Conventions for Referencing Manual Pages

The *BCI Driver Reference Manual*, the most closely related document to the *BCI Driver Development Guide*, is divided into four, alphabetically-arranged reference manual sections that provide specific information (routines, functions, and data structures) for driver writers:

D2X describes the system entry point routines that comprise the driver code.

D3X describes the kernel functions that are used in BCI driver code. Whereas user-level code uses system calls and library routines, driver code uses the kernel functions listed here.

D4X describes the kernel data structures that BCI drivers interface.

D8X describes the standard library functions used to write a diagnostics file for a 3B2 computer custom feature card. This section is also applicable to the 3B4000 ACP.

Throughout the *BCI Driver Development Guide* are references to the *BCI Driver Reference Manual*. Routines, functions, structures, and commands covered in the *BCI Driver Reference Manual* are used in this text with a reference to the appropriate *BCI Driver Reference Manual* section number. For example, **open**(D2X) refers to the driver entry point routine **open** page. The D in the (D2X) reference indicates that the routine, function, structure, or command is covered in the *BCI Driver Reference Manual*. The number following the D indicates the section number. For example, **open**(D2X) refers to the driver entry point **open** page, which is in Section 2 of the *BCI Driver Reference Manual*. If a routine, function, structure, or comment is in a UNIX System V Reference Manual, the section number alone appears in parenthesis. For example, the **open**(2) system call reference page is in Section 2 of the *UNIX System V Programmer's Reference Manual*.

See the introduction to any driver reference manual for a full explanation of the section numbers in the reference manuals for other driver interfaces.

Path Name Conventions

This document is designed to be applicable for 3B computers. Differences among machines are documented where appropriate. Because of the nature of the multiprocessing 3B4000 computer, it must be set up a little differently from the uniprocessing systems (such as the 3B2 or SBC computers). One of the most apparent places this shows up is in the paths to various files and directories mentioned in this document. Whenever you see a path name specified, it is the path name of a

uniprocessing UNIX system. For the multiprocessing 3B4000 computer, you can assume that the path name is the same for the multiprocessing host or that this path name is prefaced by *adj/pe#/* where # stands for the adjunt processor number. For example:

/etc/master.d directory means:
on a uniprocessing system: */etc/master.d*
on the 3B4000 adjunts: */adj/pe#/etc/master.d*

u t s

The UNIX system convention stores operating system and driver source code in subdirectories under the */usr/src/uts* directory. To support cross-environment development (developing software for one system on a different system), the *uts* directory has subdirectories that specify the system name, with each UNIX system kernel (3B2, 3B15, SBC, and so forth) having a unique name for this directory. In addition, each type of 3B4000 adjunct processing element has its own *uts* subdirectory where operating system and driver code for that type of adjunct processor is stored.

Table 1−2 Location of *uts* Subdirectories

Computer	Kernel Source Code
SBC	/usr/src/uts/3b2100vme
3B2	/usr/src/uts/3b2
3B15	/usr/src/uts/3b15 /usr/src/uts/com
3B4000 MP	/usr/src/uts/3b15 /usr/src/uts/com
3B4000 ACP	/usr/src/uts/acp
3B4000 EADP	/usr/src/uts/eadp
3B4000 ADP	/usr/src/uts/adp

A file's exact location in these directories may vary between releases so be sure to consult the documentation supplied with your computer.

Related Learning Support Materials

AT&T offers a number of documents and courses to support users of our systems. For a complete listing of available documents and courses, see:

AT&T Computer Systems Documentation Catalog (300-000)
AT&T Computer Systems Education Catalog (300-002)

The following list highlights documents and courses that are of particular interest to device driver writers. Most documents listed here are available from the AT&T Customer Information Center (CuIC). Documents available from CuIC have an ordering code number, which is the six-digit number in parentheses following the document title. In addition to AT&T documents, the following list includes some commercially-available documents that are also relevant.

This document is the *AT&T UNIX System V Block/Character Interface (BCI) Driver Development Guide*. Its ordering code number is 307-191.

Related Documents

Driver Development

UNIX System V Block/Character Interface (BCI) Driver Reference Manual (307-192)
includes reference material to be used in conjunction with this manual. Describes driver entry point routines (Section D2X), kernel-level functions used in BCI drivers (Section D3X), data structures accessed by BCI drivers (Section D4X), and standard library functions used to write a diagnostics file for a 3B2 computer custom feature card (D8X).

UNIX System V Portable Driver Interface (PDI) Driver Design Reference Manual (305-014)
defines the kernel functions, routines, and data structures used for developing block drivers that adhere to the UNIX System V, Release 3, Portable Driver Interface.

UNIX System V SCSI Driver Interface (SDI) Driver Design Reference Manual (305-009)
defines the input/output controls, kernel functions, and data structures used for developing target drivers to access a SCSI device.

STREAMS

UNIX System V STREAMS Primer (307-299)
provides an introduction to using the STREAMS driver interface and accessing STREAMS devices from user-level code.

UNIX System V STREAMS Programmer's Guide (307-227)
tells how to write drivers and access devices that use the STREAMS driver interface for character access.

C Programming Language and General Programming

Bentley, Jon Louis, *Writing Efficient Programs* (320-004), NJ, Prentice-Hall, 1982.
gives suggestions for coding practices that improve program performance. Many of these ideas can be applied to driver code.

Kernighan, B. and D. Ritchie, *C Programming Language,* Edition 1 (307-136), NJ, Prentice-Hall, 1978. defines the functions, structures, and interfaces that comprise the C programming language in different UNIX system environments. A short tutorial is included.

Lapin, J. E., *Portable C and UNIX System Programming*, NJ, Prentice-Hall, 1987
discusses how to maximize the portability of C language programs.

UNIX System V Network Programmer's Guide (307-230)
provides detailed information, with examples, on the Section 3N library that comprises the UNIX system Transport Level Interface (TLI).

UNIX System V Programmer's Guide (307-225)
includes instructions on using a number of UNIX system utilities, including **make** and the Source Code Control System (SCCS).

Assembly Language

AT&T 3B2/3B5/3B15 Computers Assembly Language Programming Manual (305-000)
a description of the assembly language instructions used by most AT&T computers.

WE 32100 Microprocessor Information Manual, Maxicomputing in Microspace (307-730)
introduces the WE 32100 microprocessor and summarizes its available support products.

Operating System

Bach, Maurice J., *Design of the UNIX Operating System* (320-044), NJ, Prentice-Hall, 1986
discusses the internals of the UNIX operating system, including an explanation of how drivers relate to the rest of the kernel.

UNIX System V Reference Manuals (see the table below for ordering numbers
the standard reference materials for various releases of the UNIX System V operating system. This information is divided between three books, published separately for each system.

System Administrator's Reference Manual
administrative commands (Section 1M), special device files (Section 7), and system-specific maintenance commands (Section 8).

Programmer's Reference Manual
programming commands (Section 1), system calls (Section 2), library routines (Section 3), file formats (Section 4), and miscellaneous information (Section 5)

User's Reference Manual
all UNIX system user-level commands (Section 1)

Table 1-3 gives the select codes for the UNIX System V reference manuals that are published for each AT&T computer covered in this documentation. . .

Table 1−3 Reference Manual Select Codes

Computer System	UNIX System V Release	Reference Manual		
		Administrator's	Programmer's	User's
SBC	3.1	307-056	307-053	307-057
3B2	3.1	305-570	307-013	307-012
3B15	3.1.1	305-205	305-212	305-205 †
3B4000	3.1.1	305-205	305-212	305-205 †

† For the 3B15 and 3B4000 computers, UNIX System V Release 3.1.1, the *User's* and *Administrator's Reference Manuals* are published as one volume.

Single Board Computer (SBC)

UNIX System V/VME System Builder's Reference Guide (307-068)
gives important information needed to write drivers for the SBC computer, including the firmware interface, system operation, trouble shooting, and diagnostics.

Software Packaging

UNIX System V Application Software Packaging Guide (305-001)
a cross product book describing how to write the INSTALL and DEINSTALL scripts necessary to install a driver (or other software) under the System Administration utility.

How to Order Documents

To order the documents mentioned above

- within the continental United States, call 1 (800) 432-6600
- outside the continental United States, call 1 (317) 352-8556
- in Canada, call 1 (800) 255-1242

Related Training

Driver Development

UNIX System V Release 2 Device Drivers (UC/CS1010)
explores device driver mechanisms, operating system supplied functions, and example device driver source code.

UNIX System V Release 3 Device Drivers (UC/CS1041)
explores device driver mechanisms, operating system supplied functions, and example device driver source code.

C Programming

C Language for Experienced Programmers (UC/CS1001)
covers all constructs in C language.

Internal UNIX System Calls and Libraries Using C Language (UC/CS1011)
Introduces the techniques used to write C language programs. Topics include the execution environment, memory management, input/output, record and file locking, process generation, and interprocess communication (IPC).

Operating System

Concepts of UNIX System Internals (CS1019)
overviews the main structures and concepts used internally by the UNIX operating system.

UNIX System V Release 2 Internals (UC/CS1012)
an in-depth look at the UNIX System V Release 2 internal structures, concepts, and source code.

UNIX System V Release 3 Internals (UC/CS1042)
an in-depth look at the UNIX System V Release 3 internal structures, concepts, and source code.

How to Receive Training Information

To receive information (such as registration information, schedules and price lists, or ordering instructions) about UNIX system or AT&T computer training

- within the continental United States, call 1 (800) 247-1212
- outside the continental United States, call 1 (201) 953-7554

How to Make Comments About This Document

Although AT&T has tried to make this document fit your needs, we are interested in your suggestions to improve this document. Comments cards have been provided in the front of the document for your use. If the comment cards have been removed from this document, or you have more detailed comments you would like to give us, please send the name of this document and your comments to:

AT&T
4513 Western Avenue
Lisle, IL 60532
Attn: District Manager--Documentation

Chapter 2: Introduction to UNIX Device Drivers

Contents

Introduction

This chapter introduces most of the basic concepts a programmer should understand before attempting to write a UNIX system device driver. Each major topic is covered more fully in later chapters; experienced driver writers may wish to turn directly to these detailed discussions. This chapter gives an experienced C programmer an overview of how to write a device driver, by showing

- how device drivers resemble and differ from application programs
- the different types of device drivers, and what they have in common with each other
- what files must be created or modified so that a driver may be installed on a system
- two example drivers that illustrate the main components of most drivers and what those components typically do
- some guidelines for developing a driver

What is a Device Driver?

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

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

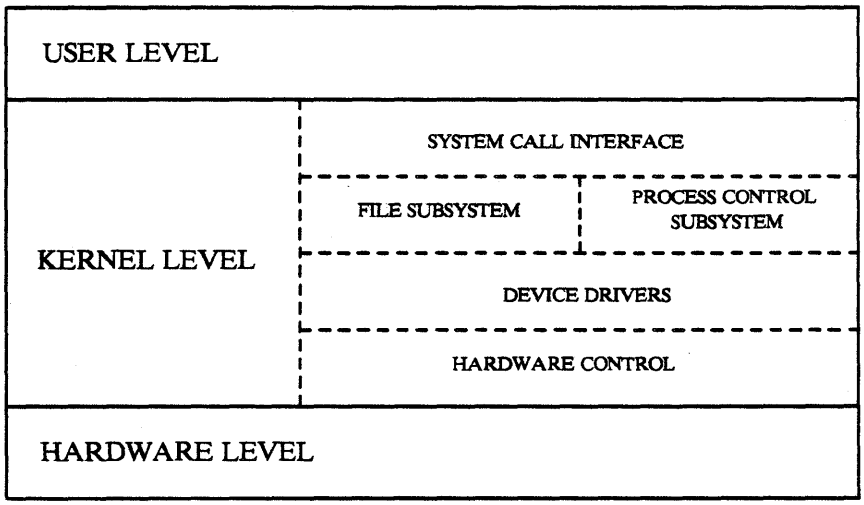


Figure 2–1 Driver Placement in the Kernel

Figure 2-1 shows how a driver provides a link between the user level and the hardware level. By issuing system calls from the user level, a program accesses the file and process control subsystems, which, in turn, access the device driver. The driver provides and manages a path for the data to or from the hardware device, and services interrupts issued by the device's controller.

Application Programs vs. Drivers

This book is intended for experienced C programmers. All code examples are in the C language, and it is quite possible to write your entire driver in C. However, there are some major differences between writing a device driver and writing a program designed to execute at the user level. This section reviews some of those differences and introduces some of the system facilities used in driver development.

Structure

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

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

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

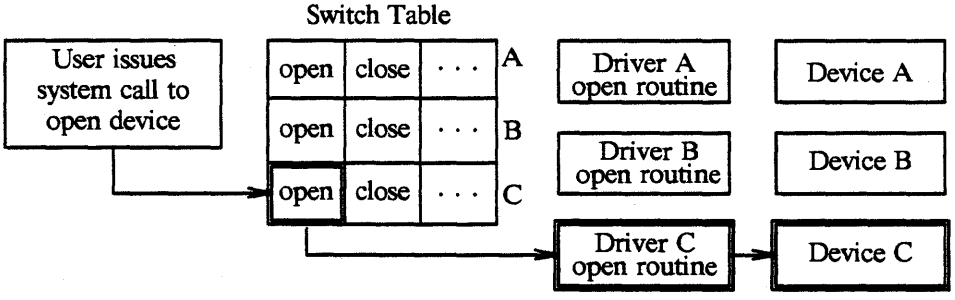


Figure 2–2 How Driver Routines Are Called

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

Parallel Execution

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

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

Interrupts

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

The processing needed to handle hardware interrupts is another of the major differences between drivers and application programs. Later in this chapter, a simplified model of an interactive terminal driver is given that a describes how a driver synchronizes its data transfer functions with its response to hardware interrupts. Chapter 9, "Synchronizing Hardware and Software Events," discusses how data movement is synchronized, and Chapter 10, "Interrupt Routines," covers interrupts in greater detail.

Driver as Part of the Kernel

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

But drivers can and do have much greater impact on the kernel. Inefficient driver code can severely degrade overall performance, and driver errors can corrupt or bring down the system. For this reason, testing and debugging driver code is particularly challenging, and must be done carefully. Chapter 13, "Testing and Debugging the Driver," discusses the facilities available for finding drivers errors, as well as some of the special problems that are encountered when testing driver code.

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

- A number of header files, used to declare data types, initialize constants, and define system structures, must be included in the driver source code. The exact list of header files varies from driver to driver. See Chapter 4, "Header Files and Data Structures," for more details.

- Various structure members and device registers must be read or written, and usually some system buffering structure must be used. Many of the functions included in the interface are designed to be used with these structures. These structures are explained in Section D4X of the *BCI Driver Reference Manual*.

- Drivers have no access to standard C library routines. Yet, the routines included in the block and character interface represent a kind of library and provide some functions similar to those found in the standard C library. On the other hand, the interface also provides many functions that are unlike standard C library functions. See Section D3X of the *BCI Driver Reference Manual* for complete explanations of the interface routines.

- Drivers cannot use floating point arithmetic.

Types of Devices

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

Hardware Devices

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

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

Software Devices

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

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

The Block and Character Interface

An *interface* is the set of structures, routines, and optional functions used to implement a device driver.

Block and character are the two interfaces described in this book, and correspond to the two basic ways drivers move data. Block drivers, using the system buffer cache, are normally written for disk drives and any mass storage devices capable of handling data in blocks. Character drivers, the typical choice for interactive terminals, are normally written for devices that send and receive information one character at a time.

It is the individual device, not the device type, that determines whether a driver should be the block or character type. For example, one printer, capable of data buffering, may be a candidate for a block driver, while another printer may need a character driver.

Furthermore, one device may have more than one interface. A disk drive may have both a block and character interface. This situation is explained in Chapter 6, "Input/Output Operations."

Alternative Interfaces

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

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

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

See the *UNIX System V STREAMS Primer* and Chapters 9 and 10 of the *UNIX System V STREAMS Programmer's Guide* for *STREAMS* driver details.

AT&T has defined an interface, called the Portable Driver Interface (PDI). The PDI is a collection of driver routines, kernel functions, and data structures that provide a standard interface for writing UNIX System V block drivers. PDI is usable on all 3B2, 3B15, and 3B4000 computers runing UNIX System V Releases 2.0.5, 3.0, 3.1, or later. For more information about our PDI documentation, see Chapter 1, "Related Documents."

Small Computer System Interface (SCSI) devices use a collection of machine-independent input/output controls, functions, and data structures, that provide a standard interface (called SCSI Driver Interface (SDI)) for writing SCSI target drivers to access a SCSI device. For more information about our SDI documentation, see Chapter 1, "Related Documents."

Driver Environment

A device driver is added to a working UNIX system in three basic steps including

1. **Configuration Preparation** -- Involves modifying or creating system files on an active system. During the preparation phase, a bootable object file is created with either the **drvinstall**(1M) or **mkboot**(1M).

2. **Configuration** -- Invoked by shutting down and rebooting the system. The system uses information from the modified system files to include entries for the new driver in system structures.

3. **Initialization** -- The driver itself is then initialized as part of overall system initialization.

The major steps are reviewed here; Chapter 12, "Installation," gives more details about how drivers are configured and installed, and Chapter 5, "System and Driver Initialization," discusses system initialization.

Configuration

For a driver to be recognized as part of the UNIX system, information about what type of driver it is, where its object code resides, what its interrupt priority level will be, and so on, must be stored in appropriate files. Chapter 5, "System and Driver Initialization," summarizes what information is required, and how it is used in configuration.

The following are used when configuring a driver into the system:

/etc/master.d	This directory contains the master files. A master file supplies information to the system initialization software to describe the attributes of a driver. There is one master file for each driver on the system.
/etc/system	This file contains entries for each driver and indicates to the system initialization software whether a driver is to be included or excluded during configuration.
/dev	This directory contains special device files. A device file establishes a link between a driver and a device.
/boot	This directory contains bootable object files that are used to create a new version of the UNIX operating system when the processor is booted.

Driver Structures

The master file is the source of some of the more important information used by the configuration process. From information provided there, several system structures are built to make drivers part of a bootable system. Three of them are of particular interest

- The MAJOR and MINOR tables contain numbers used by the kernel to identify drivers. The major number identifies the driver, and the minor number identifies the subdevice. A subdevice might be one of several disks controlled by a single driver or one of many terminals. Usually, the minor number is passed as an argument to the driver to identify the particular subdevice.
- Two switch tables (`bdevsw`(D4X) for block and `cdevsw`(D4X) for character drivers) contain the starting addresses for the entry point routines for all installed drivers.
- Two other tables (**io_init** and **io_start**) are built to hold the initialization routines.

Driver Prefix

Every driver's master file contains information, used during system configuration, about the specific attributes of drivers. One of the fields in the master file is the *prefix* (a string of up to four characters) added to generic routine names (such as "init," "open," and so on). For example, a RAM disk driver may have been given a prefix of "ram_" resulting in routines named "ram_open," "ram_init" and so on.

During configuration, the system looks in the master file for the prefix, and then looks for the entry point routines with matching prefixes. The addresses of these routines are loaded into the switch tables (and, in the case of the **init**(D2X) routine, into the **io_init** table).

Initialization

Not all drivers have **init**(D2X) routines; some have nothing to initialize and others defer initialization to the **open**(D2X) routine. In most cases, it doesn't matter if variables are zeroed in an **init** or an **open** routine. On the other hand, the system should be informed at the time of initialization if, for example, a disk drive is off-line.

Software drivers typically have little to initialize because no hardware is involved. In fact, some software drivers have completely empty **init** routines. Memory may be allocated as a simple two-dimensional array in the **open** routine. But even if no **init** routine is needed, the driver must have an entry point routine in the switch table.

In the following pseudo-code for a software driver, initialization processing required is minimal. Some memory must be allocated and initialized, and a warning must be issued if the allocation fails.

The numbers in parentheses (before the lines of pseudo-code) are referenced by the section headers below, to indicate which line is being explained in that section. In most cases, an actual code fragment from a working driver is included to help illustrate the concept.

```
(1)     include header files


          init(dev)

(2)             if (memory can be allocated)
                allocate memory
                initialize memory
(3)             print informational message
              else
                print warning message
```

The standard library of C functions cannot be used in driver code. However, most of a driver's processing is performed by the functions described in Section D3X of the *BCI Driver Reference Manual*. To use the interface effectively, it is important that you become familiar with what these functions can do. Some of them are introduced in the discussion of the sample drivers, but many more are available and are illustrated both in this document and in the *BCI Driver Reference Manual*.

Driver Header Files (1)

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

Other required header files are mentioned as needed, and a complete list of available header files appears in Appendix B, "Writing 3B2 Computer Diagnostics Files." Most drivers will need to include a minimum of 5 to 10 header files and some may have more than 20.

Memory Allocation (2)

The function used to allocate memory is **kseg**(D3X), shown in the *BCI Driver Reference Manual.* The reference page shows that **kseg** accepts as an argument the number of pages to be allocated (up to 64), and that the pages are segment-aligned and cannot be swapped out. The **kseg** manual page also tells you what conditions must exist for the allocation to succeed, how different types of failures are handled, and which header files must be used.

Messages (3)

Another useful library function is **cmn_err**(D3X). The **printf**(3S) library function cannot be used in driver code; instead, the function **cmn_err** is used for all types of messages, from the merely informational to those reporting severe errors. The first argument to this function is a constant used to indicate the severity level, the second is the text of the message, and the third is an optional variable. For example, the following statement could be used to report why the initialization failed:

```
cmn_err(CE_WARN,"init: kseg cannot allocate %d buffers", BUFS);
```

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

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

Other init Responsibilities

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

- Check to see if the devices under the control of the driver are actually on-line.
- Check for the correct number of subdevices.
- Set each device's interrupt vector to correspond to the system's interrupt vector table.
- Set the virtual-to-physical address translation.

- Set device-specific parameters to default values. These parameters include values for the number of tracks, cylinders and sectors.
- Download executable code to the controller. Controllers for many devices have their own processors and memory and are referred to as *intelligent devices*. The executable code downloaded to the controller is called *pumpcode*.

See Appendix E, "Sample Block Driver," for a detailed explanation of actual code for a disk driver.

Example Block Driver

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

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

The work of the base level of a RAM disk driver is to open a file system, provide access to it, and close it when necessary. The entry point routines required for these activities are **open**(D2X), **strategy**(D2X) and **close**(D2X). The only other part of the RAM disk driver is the initialization routine (**init**(D2X)), illustrated in the previous section.

Each routine is illustrated (with pseudo-code) in the pages that follow. After the pseudo-code is a brief discussion of every line of the pseudo program. Some of these is include actual code fragments from a working driver.

Base-Level Operation

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

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

The open Routine

When a user process issues an **open**(2) system call, the file to be opened is most often a regular file. The purpose is generally to read or write text or data. However, the driver **open**(D2X) routine is opening the device, which looks like a file on a UNIX system. Chapter 3, "Drivers in the UNIX Operating System," explains these files in more detail, but two points are important here

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

Finally, when the open routine is called, it is passed the device number and the flags indicating the type of open (read only, create new file, and so on).

```
include header files

   open(device number, flags)

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

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

Validating the Minor Device Number

The device number is a two-byte quantity containing both the major number (identifying the driver) and the minor number (identifying the subdevice). By the time the **open**(D2X) routine has been called, the major number has already been used as an index into the switch table to select the driver. The device number is passed to the **open** routine as an argument and the minor portion of it is extracted with the **minor**(D3X) macro.

```
if (minor(dev) > MAXDEV)
```

An error results if an invalid minor number, a number greater than the constant MAXDEV (declared in the driver code), is detected.

Writing Errors to the user Structure

When a driver needs to report an error to the user, the usual method is to set the **u.u_error** member of the `user` structure, described in Section D4X of the *BCI Driver Reference Manual*. For example, if the minor number (extracted with the **minor** macro) is found to be out of range, the RAM driver uses the constant ENXIO to indicate a non-existent device.

```
u.u_error = ENXIO;
```

The available error constants are defined in *errno.h* and the `user` structure is defined in *user.h*.

Setting Up a Buffer

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

The section called "The strategy Routine" explains how the driver reads and writes blocks. This section introduces the buffer header, the part of the buffer structure used to identify where the data came from. The structure is called `buf`(D4X), and is defined in the file *buf.h*.

This RAM driver contains a file system and so must have access to file system information stored in the superblock. To make this possible, the **open** routine declares a pointer to a `buf` structure, loads some buffer header values, and then calls the driver **strategy** routine to read the superblock. (Notice that it is possible for one entry point routine to call another.) If the read fails, the error is reported by writing to the **u.u_error** member of the `user` structure, as shown in the **init** routine.

The Buffer Header

The buffer cache contains buffers of data belonging to many devices. The buffer header contains information used to keep them straight. The following header members must be set before reading the superblock. For a complete description of the `buf` structure, see the structures section (Section 4) of the *BCI Driver Reference Manual.*

- **b_dev.** The device number. (A composite value, made up of both the major and minor number.) It is used to identify the RAM device.
- **b_bcount.** The number of bytes to be transferred. When reading the superblock, a full block is to be read, so this member is set to 1024 for this system.
- **b_blkno.** The device's block number, set to the superblock.
- **b_error.** The **open** routine sets the error number to zero, before the first read. Later, the **strategy** routine sets this member on I/O failure.
- **b_flags.** Values are ORed into this member (allowing more than one value to be on at a time). For example, two values are set before a read of the superblock

```
bp->b_flags |= B_BUSY | B_READ
```

B_BUSY indicates the buffer is in use; B_READ determines the direction of data transfer (from the device to memory). A write is indicated by B_READ not being set.

After the buffer header values have been loaded, the driver's own **strategy** routine is called, with a pointer to the buffer header as an argument (*bp*). After the read is attempted, the **b_flags** member is tested to see if an error has occurred.

```
if (bp->b_flags & B_ERROR)
```

Other open Routine Responsibilities

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

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

The Strategy Routine

As shown in the previous section, the **strategy**(D2X) routine is called from the **open** routine to read the superblock. More often, **strategy** is called in response to a system I/O request. That is the main work of the driver, and **strategy** is the routine that does it.

For now, it is not necessary to understand in detail how the kernel manages the buffer cache. (More information about that is provided in Chapter 6, "Input/Output Operations.") To transfer data, the **strategy** routine is passed a pointer to a buffer header in the system buffer cache. The buffer header contains all necessary information about the source and destination of the transfer and how many bytes will be moved.

```
include header files

     strategy(bp)

    if (block number is out of range)
       write error to user structure
       return

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

    call iodone
    return
```

Check for Valid Block

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

```
if (bp->b_blkno < 0 || bp->b_blkno >= RAMBLKS)
```

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

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

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

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

Reading and Writing Data

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

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

The iodone Function

When the data transfer is complete, the **strategy** routine calls the **iodone**(D3X) function. Hardware drivers use **iodone** to awaken sleeping processes, which is not required for pseudo-devices. The RAM driver uses this function to release the buffer block and to set the **b_flags** member to B_DONE. The **iodone** function is called with a single argument, the pointer to the buffer header.

```
iodone(bp);
```

The close Routine

Many drivers (even hardware drivers) will have empty **close**(D2X) routines. Even though it does nothing, the address of the empty routine is entered into the switch table.

```
close( )
{
}
```

If not empty, a **close** routine may be responsible for unlocking the device (if locked by the **open**(D2X) routine), flushing buffers, making sure the device does not contain a mounted file system, and reinitializing its data structures.

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

For more information, see the reference page for for **close** in Section D2X on the *BCI Driver Reference Manual*.

Example Character Driver

Character drivers are used for data transfers where it is not possible to organized the data into blocks. Interactive terminals and networks are the most common devices of this type. Like block drivers, character drivers use a switch table (`cdevsw` instead of `bdevsw`) to store base level routine entry points, and have **init, open**, and **close** routines. But unlike block drivers, character drivers have **read** and **write** routines instead of **strategy**, and can also include a general purpose I/O control (**ioctl**) routine for changing terminal settings, for example.

The terminal driver described in this section demonstrates these and other features peculiar to character drivers, along with some of the features common to both block and character hardware drivers that are not part of the RAM disk driver. The most important of these is the code required to handle interrupts.

Line Disciplines

The processing necessary to drive an interactive terminal is more complicated than for the RAM disk driver, but there are also more standard routines supplied as aids. Among these are a group of routines known collectively as a *line discipline*.

While it is possible to write your own line discipline and configure it into the system, a standard line discipline (called line discipline zero) is suitable for most character drivers.

The routines of the line discipline correspond to the routines of the driver, and like a driver, are accessed through a switch table (`linesw`). Typically, a terminal driver routine performs some driver-specific processing and then calls the corresponding line discipline routine.

Another group of standard routines are known as the TTY subsystem. These are part of the character interface, and each has a page in Section D3X of the *BCI Driver Reference Manual*. Their use is demonstrated in the example pseudo-code driver that follows, and more fully in Chapter 7, "Drivers in the TTY Subsystem."

The open Routine

The most important component of the TTY subsystem is the `tty` structure. There is one instance of this structure for each configured port, providing a standard method for storing most of the information needed by the driver. Two members of the `tty` structure are used by the **open**(D2X) routine.

- **t_line**, which identifies the line discipline used by this driver.
- **t_state**, which is a set of 16 flags used to describe the current state of the device and the driver.

(For a complete description of this structure, see Section D4X of the *BCI Driver Reference Manual.*)

The use of these and other members of the `tty` structure are described as they are used.

```
include header files
declare structure for device registers

   open( )

    get device registers
    get port number

    if (device not open)
      initialize tty structure

    if (physical connection not made)
      wait for connection

    call line discipline open
```

Header Files

Except for *buf.h,* all of the header files mentioned in the block driver example must also be included in the terminal driver. In addition, include the *tty.h* file, which declares the `tty` structure. The line discipline switch table (`linesw`) is defined in *conf.h.*

Declare Device Register Structure

Device registers are special memory locations by which the driver communicates with the device. The structure includes four main members

- *control* word used to pass the type of parity, number of stop bits and other information.
- *status* word used to make the status of the device (sending, receiving, and so on) known to the driver.
- *receive* character, to hold the last character received from the device.
- *transmit* character, to hold the last character transmitted to the device.

Get Device Registers

The device registers are accessed by using the minor device number to index an externally declared array.

```
*rp = &addr[minor(dev) >> 3];
```

Get Port Number

Like the device registers, the port number uses the minor device number (ANDed with the constant "7" for this controller) to find the correct value.

```
port = minor(dev) & 0x07;
```

Initialize tty Structure

Because this driver uses line discipline zero, a standard TTY subsystem function can be used to initialize the port's `tty` structure. The function **ttinit** sets **t_line** and several other values to zero, and loads a default set of control characters into a character array, **t_cc[]**. The characters loaded are delete, quit, erase, kill, and end of file.

The function is called with a pointer to the `tty` structure as an argument

```
ttinit(tp);
```

Wait for Physical Connection

The **t_state** member of the `tty` structure is used to test the carrier-present signal. If the device is not found to be on-line, the WOPEN bit in the same member is set.

```
while(!(tp->state&CARR_ON))
    tp->t_state |= WOPEN;
```

The sleep Function

While waiting to detect a physical connection, the **open**(D2X) routine calls the **sleep**(D3X) function. This function is used to suspend execution of the driver when it is called and wait for some event to occur. Most often, the event is the completion of a data transfer, but here it is waiting for a line to be activated. In either case, the routine sleeps until it receives a **wakeup**(D3X) call from the interrupt routine.

Many sections of driver code use the **sleep** function and a variety of hardware events are detected by the interrupt routine. The first argument to both the **sleep** and **wakeup** functions (sometimes called an *event*) is an address used to identify a hardware event and match a **sleep** and **wakeup** call.

The address chosen in this case is one of the members of this port's `tty` structure. By choosing a memory address allocated to this port's invocation of the driver, conflict with other calls to **sleep** and **wakeup** can be avoided.

The second argument to the **sleep** function is the priority level, which is discussed later.

```
sleep((caddr_t)&tp->t_canq, TTIPRI);
```

In at least one place in the interrupt routine (there may be more), the above **sleep** call has a corresponding **wakeup** call to resume execution.

```
wakeup((caddr_t)&tp->t_canq);
tp->t_state |= CARR_ON;
```

Call Line Discipline

After the driver-specific processing is complete, the line discipline **open**(D2X) routine is called to establish the logical data connection.

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

Among other functions,the line discipline open routine allocates a buffer to receive characters (the **t_rbuf** member of the `tty` structure) and calls the drivers **proc(D2X)** routine. Both of these are discussed later in this section.

The close Routine

The driver's **close**(D2X) routine does nothing more than call the line discipline **close** routine. The line discipline takes care of both the logical and physical disconnection, and clearing and deallocating buffers. Other driver **close** routines might have to reset driver structure members and perform other clean-up.

```
close( )

      call line discipline close
```

The read Routine

The line discipline routine normally does everything the driver **read**(D2X) routine is required to do. The line discipline mainly takes the data from the raw input queue, and calls the **canon**(D3X) function to process ERASE and other non-data characters.

```
read( )

      call line discipline read
```

The proc Routine

This routine is called both directly by driver routines and indirectly by some of the line discipline routines. To take advantage of using line discipline calls, the device-specific processing must be isolated in a **proc**(D2X) routine and made accessible to the line discipline.

The **proc** routine is passed a pointer to a `tty` structure and a command to be processed. The driver **open** routine, for example, calls **proc** with the command set to T_INPUT, to prepare the device to receive input. The driver **write** routine, on the other hand, calls **proc** indirectly through the line discipline **write** routine (with a command value of T_OUTPUT). (See Section D4X of the *BCI Driver Reference Manual* for more information about the commands a **proc** routine must be able to process.)

The write Routine

The line discipline **write** routine is responsible for some processing similar to the canonical processing done by the **read** routine. Tab characters are expanded to the correct number of blanks and delay routines accommodate newline and backspace characters.

```
write( )

      call line discipline write
```

I/O Controls (The ioctl Routine)

A terminal driver has an **ioctl**(D2X) entry point routine to respond to user requests to change terminal settings. (The request is expressed as an **ioctl**(2) system call, but may be indirectly called by the **stty**(1) command.)

```
ioctl(dev, cmd, arg, flags)

    get tty structure

    if (tty structure has no errors)
      get device registers
      change terminal settings
```

Get tty Structure

The first argument to the routine is the device number, and it is used to set a pointer to the instance of the `tty` structure for this port.

```
device = minor(dev);
tp = &tty[device];
```

Check tty Structure for Errors

Next, the kernel function **ttiocom**(D3X) is called and its return value is tested. A non-zero return value indicates no errors have been detected. At the same time, the *cmd* argument is passed to the **ttiocom** function to set parameters in the `tty` structure.

```
if(ttiocom(tp, cmd, arg, flags))
```

Get Device Registers

Changing the `tty` structure does not change the terminal settings. The device is accessed only through the device registers.

```
rp = &addr[minor(dev) >> 3];
```

For portability, the code for setting terminal parameters is isolated in a subordinate routine and is specific to the hardware involved.

```
param(dev);
```

Interrupt Routines

The terminal driver has to respond to interrupts caused by several different sources, including the following:

- the terminal user has pressed a quit, delete, control-s or some other key
- the terminal is ready for output
- data transfer is complete
- some kind of error has been detected

To service a variety of interrupts correctly, the interrupt routine selects from a list of cases by interrupt opcode, a value passed to the routine. A typical section will perform one or more of these services

- set flags in the **t_state** member of the `tty` structure
- call a line discipline routine
- call the **proc** routine
- flush buffers
- set flags to reflect the state of the board
- call the **wakeup** function

Setting Priority Levels

Some data structures, such as `tty`, can be modified by both base-level and interrupt-level routines. Because interrupts can occur at any time, precautions must be taken to postpone an interrupt at places in the code where common structures may be modified. These areas of driver code are called *critical sections*.

A set of functions are used to temporarily raise a processor priority level and then return it to the previous level after the critical section has finished executing. The **spl7** and **splhi** functions set the priority level to 15, preventing all interrupts. (See the **spl***n*(D3X) entry in the *BCI Driver Reference Manual* for the uses of each level. See Chapter 9, "Synchronizing Hardware and Software Events.")

Normally, a critical section of code is protected by saving the old priority level and then restoring it with the **splx** function, as shown.

```
oldlevel = spl4();
  . . .
```

critical section

```
  . . .
splx(oldlevel);
```

Driver Development

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

Basic Steps for Creating a Driver

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

Preparation

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

Implementation

- Write and install a minimal driver. It is very helpful to test driver code from the earliest stages, and to verify that it can be installed. A minimal driver might be one that simply uses the **cmn_err** function to send a "hello, world" message to the system console. See Chapter 12, "Installation," for a detailed guide to driver installation.
- Write base-level routines before interrupt-level routines.

- If applicable to the device, write and test any associated firmware driver.
- Develop utilities such as disk formatting, network administration, and diagnostic programs at the same time as the driver.

Follow-up

- As much as possible, use the testing phase to create error conditions that exercise the driver's ability to handle them.
- Evaluate the driver's performance both in isolation and in a production environment where other drivers are installed. Regression testing should be performed to ensure that a new device driver does not affect other system functionality.
- Make sure documents affected by the creation of the driver are updated. These may include operator and diagnostic manuals and sales or ordering information.
- If the driver is to be installed by a customer, write and test installation and deinstallation packages, as described in Chapter 16, "Packaging the Driver."

Commenting Driver Code

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

- Each file should have a comment block at the beginning, describing the type of file functions and the services they perform. List the functions that call them and the functions they call. For a hardware driver, describe the hardware, including version numbers and hardware strapping values.
- Describe each global data structure or type declared, including its possible range of values. Describe the protocol, if any, used to access it (such as flag-setting). If it is useful, describe the functions that access structures, including those that are in other files.
- Each routine should have a comment block at the beginning describing what it does, how it does it (what are the algorithms or strategy), assumptions about the environment when it is called (processor interrupt priority level, outstanding I/O jobs, and so forth), and what global variables are used.
- Each line that declares an argument to the routine should have a comment.
- Every local variable should be explained.

- Each loop or "if" test should have a comment to explain the exit condition.

Layered Structure

Hardware drivers will be easier to port and maintain if structured in layers. Separate the higher-level protocol functionality from the low-level, machine-dependent routines. The high-level sections can be readily ported, leaving only the low-level sections to be rewritten. If machine-specific code is not isolated, all code may need to be rewritten to run on another processor.

Also, when your driver accesses system structures such as the system buffer cache and the user and `proc` structures, use the standard functions included in the basic interface. Using non-standard functions with standard structures can degrade the performance of other drivers on the system and will impact portability.

Driver Functions

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

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

Utilize Board Intelligence

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

With modern intelligent devices, some of the control for a device or controller may be in code running on the controller board rather than in the driver running in kernel memory. The code for the controller board may be in firmware or may be downloaded to controller RAM, for example, at system boot time. If the device never needs to work in a non-UNIX system (firmware) mode, it is

not necessary to use firmware for anything more than diagnostics, interrupt structure, and the interface to the Equipped Device Table (EDT), discussed in Appendix A, "The Equipped Device Table (EDT)." Otherwise, to copy data to and from your device in a non-UNIX system mode, the fundamental functionality for the board must be burned in firmware. You may also want to include in firmware a basic subset of the protocol necessary to talk to the host processor directly, such as the memory management protocol. Proper use of firmware can enhance the features, portability, and performance of your device.

Pumpcode is firmware code that is stored in UNIX system files and downloaded (or "pumped") to the board during system startup. Code can be pumped by the initialization routines discussed in Chapter 5, "System and Driver Initialization," (if it is required that early), or by I/O control commands that you define as discussed in Chapter 8, "Input/Output Control (ioctl)." It is occasionally also pumped by programs called by the **init**(1M) process. For instance, on the 3B15 computer, pumpcode for the I/O Accelerator (IOA) is not sent to the board until the machine enters multiuser state.

Firmware must be coded according to the microprocessor board specifications. The */usr/include/sys/firmware.h* file defines the structures the memory board requires to communicate with the boards. In addition, the firmware board must adhere to the diagnostic interface, EDT interface, and interrupt structure for the system. Chapter 1, "About This Document," describes other documents where this information is available for the microprocessors used in the computers documented in this book. Appendixes A and B review the EDT interface and diagnostic interface, respectively.)

References

For more information on all of the driver routines mentioned in the two examples, refer to the chapters listed in Table 2-1. Reference manual pages are provided for each routine in the *BCI Driver Reference Manual*.

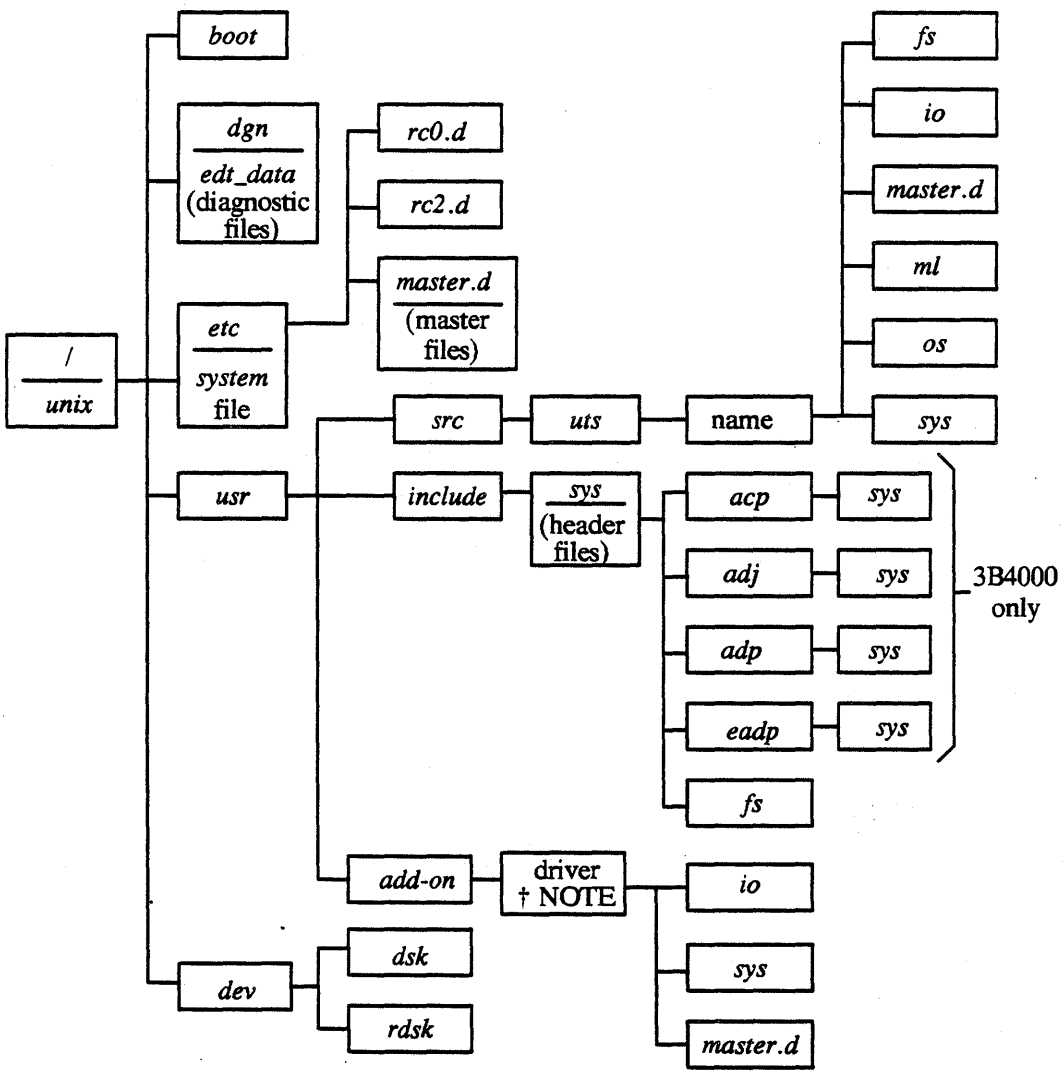
Table 2–1 Driver Entry Point Routines

Initialization Entry Points	**init(D2X)**	Chapter 5
	start(D2X)	Chapter 5
Switch Table Entry Points	**open(D2X)**	Chapter 5
	close(D2X)	Chapter 5
	read(D2X) (character-access only)	Chapter 6
	write(D2X) (character-access only)	Chapter 6
	ioctl(D2X) (character-access only)	Chapter 8
	strategy (block-access only)	Chapter 6
	print (block-access only)	Chapter 11
Interrupt Entry Points	**int(D2X)**	Chapter 10
	rint(D2X)/xint(D2X)	Chapter 10

System and Configuration Files

This section is an introduction to the basic files you need to become familiar with when configuring a driver into the UNIX operating system, such as the location of source files and the creation of a master file in the */etc/master.d* directory.

Figure 2-3 shows the files and directories used when creating or maintaining a driver.



† **NOTE:**

Name	For this type of computer
32100vme	Single Board Computer (SBC)
3b15	3B15 Computer
3b2	3B2 300/400/500/600 Family
acp	3B4000 Adjunct Communications Processor
adj	3B4000 Adjunct Processors' Common Directory
adp	3B4000 Adjunct Data Processor
com	3B4000 Master Processor and 3B15 Common Directory
eadp	3B4000 Enhanced Adjunct Data Processor

Figure 2–3 Files and Directories Used by Drivers

Of the files listed above, the following are important for system configuration:

/etc/master.d — this directory contains the master files. A master file supplies information to the system initialization software to describe the attributes of a driver.

/etc/system — this file contains entries for each driver and indicates to the system initialization software whether a driver is to be included or excluded for configuration.

/dev — this directory contains special device files. A device file establishes a link between a driver and a device.

/boot — this directory contains bootable object files that are used to create a new version of the UNIX operating system when a computer is booted.

Certain system files and directories must be informed of your driver; depending on the initialization that is required, you may need to add entries to others.

Table 2-2 lists the files and directories you may need to modify for your driver.

Table 2–2 System Files Used By Drivers

	System File	Purpose
	/etc/system	Controls construction of the operating system
† †	*/dgn/name* */dgn/X.name* */etc/scsi.d/name* */etc/scsi.d/X.name* */adj/peNNN/dgn/name* */adj/peNNN/dgn/X.name*	Diagnostic code: (3B4000 MP equipped with SCSI) (3B4000 ACP, NNN is the processor element number)
*	*/etc/master.d/**	Configuration information for the device or module
*	*/boot/**	Compiled driver, processed with **mkboot**(1M)
	*/lib/pump/**	SBC/3B2 computers (and 3B4000 ACP) pumpcode
	*/lib/bootpump.d/**	3B15/3B4000 computers pumpcode
	*/etc/brc.d/**	Scripts to be executed before those in */etc/rc.d*
	*/etc/rc.d/** */etc/bcheckrc*	Scripts to be executed when system goes to multiuser state (3B2 computers)
	/etc/rc0	Script to be executed at shutdown

* indicates an element that must be updated for all drivers.
/dgn files must be present for new hardware boards (cards) and for all SBC drivers. For SBC drivers, you should link a file with the same name as your driver in all upper case to the null diagnostics file and to the corresponding *X.* diagnostics files. These files are required before your system can be booted. Refer to Appendix B, "Writing 3B2 Computer Diagnostics Files," for more information on writing a */dgn* file.

Refer to the reference manual pages in the *Programmer's Reference Manual* under **master**(4) and **system**(4) for more detailed information on the */etc/system* and master files.

These files are used for self-configuration and system initialization. Chapter 5, "System and Driver Initialization," discusses the self-configuration and system initialization processes.

Chapter 3: Drivers in the UNIX Operating System

Contents

Introduction

This chapter describes the means by which drivers are accessed by the UNIX operating system. The following subjects are discussed:

- driver initialization and driver initialization routines
- switch table entry points
- major and minor device numbers
- external and Internal major/minor number translation
- interrupt entry points

Driver Entry Points

As discussed in Chapter 2, drivers are accessed in three ways

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

When the system is initialized, several tables are created as a means for the system to enter drivers through their routines. Because the system uses these tables to determine the appropriate driver routines to activate, the routines themselves are sometimes referred to as driver *entry points*.

Each table is associated with a specific set of entry point routines. Initialization tables are associated with either **init**(D2X) or **start**(D2X) routines. System calls use a pair of switch tables whose entry points are **open**(D2X), **close**(D2X), **read**(D2X), **write**(D2X), and **ioctl**(D2X) routines for character drivers, and **open**, **close**, and **strategy**(D2X) routines for block drivers. Device interrupts are associated with their appropriate interrupt handling routine through an interrupt vector table whose entry points are either an **int**(D2X) routine, or a **rint**(D2X)/**xint**(D2X) routine pair.

The following sections discuss these system tables and their associated entry points in greater detail.

Initialization Entry Points

All driver initialization routines, either **init** or **start**, are executed during system initialization and are executed in a different order each time the system is configured. The system uses only the routines themselves and information from the driver's master file to initialize the drivers. Information such as the major/minor numbers, important when accessing driver switch table entry points, is not used to initialize a driver. The system does not differentiate between character- and block-access drivers when running the initialization routines.

The system initialization program first creates two internal tables, **io_init** and **io_start**, which it uses to list the routines that must be executed. After the system is initialized, the **io_init** and **io_start** tables are never accessed again. Not all drivers need initialization routines. A driver that does not have an **init** or **start** routine has no entry in the **io_init** or **io_start** table.

Chapter 5 describes the internals of system and driver initialization. Chapter 5 also gives guidelines for choosing and writing the type of initialization routine appropriate for your driver.

Switch Table Entry Points

Two operating system switch tables, `cdevsw`(D4X) and `bdevsw`(D4X), hold the switch table entry point routines for character and block drivers, respectively. These routines are activated by I/O system calls, as illustrated in Figure 3-1.

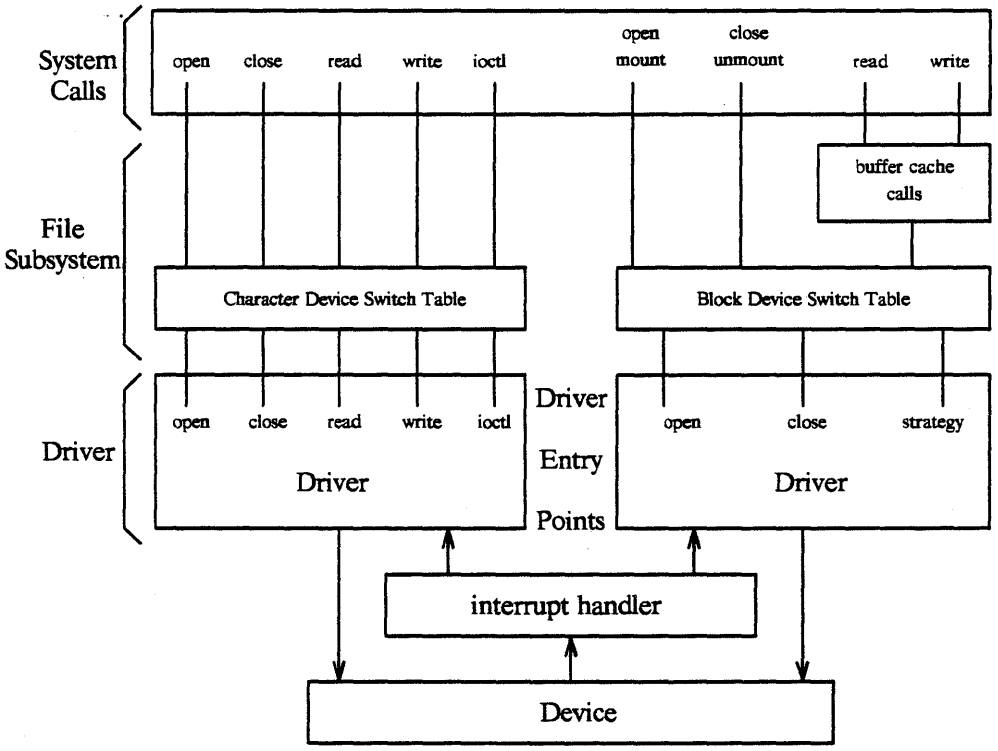


Figure 3–1 Switch Table Entry Points and System Calls

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

1 The I/O system call (**open, close, read, write,** etc.) is directed to a special device file.

2 The special device file includes the external major number for the device. Using the MAJOR translation table, the operating system finds the corresponding internal major number.

3 If the special device file is for block-access, the operating system will use the internal major number as an index into the `bdevsw` table to find the appropriate routine. For character-access, the operating system will look in the `cdevsw` table, using the same method.

4 The operating system then calls the appropriate routine.

Whenever the character-access entry points are being used, the block-access entry points are inaccessible, and vice versa. As will be discussed in Chapter 6, when doing a character-access **read** or **write** operation on a device that supports both block- and character-access, the driver calls the **strategy** routine. It calls this routine, however, as a subordinate routine to **read** or **write**, not as the `bdevsw` entry point.

Note that the `cdevsw` entry point routines for TTY drivers access subordinate routines through the `linesw` table. This is discussed in Chapter 7.

The next several sections give more details on the files and processes involved in accessing the switch table entry point routines.

Entries in Switch Tables

Figure 3-1 shows that `bdevsw` and `cdevsw` have a place for every switch table entry point that could be coded for a driver. However, not all routines are appropriate for all devices. For instance, a printer driver does not need a **read** routine. The operating system provides a place holder in the switch table for routines that are not included in the driver. Table 3-1 summarizes what the self-configuration process will enter in the switch tables for routines that are not included, and the result of attempting to call that routine.

Table 3–1 Switch Table Entries for Non-Coded Routines

Type of Driver	If you omit:	Self-config enters:	If accessed:
Any driver	**open**	**nulldev**(D3X) in `bdevsw` or `cdevsw`	no operation and no error code
Character access ("c" FLAG)	**read** **write** **ioctl**	**nodev**(D3X) in `cdevsw`	ENODEV in **u.u_error**
Block access ("b" FLAG)	**strategy** **print**	**nulldev**(D3X) in `bdevsw`	no operation and no error code

A "b" or "c" in the FLAGS column of the master file determines if entries are made in the `bdevsw` or `cdevsw` tables, regardless of what routines are coded in the driver. For instance, if you include a **strategy** routine but omit the "b" from the master file, `bdevsw` will have no entries for that device. If a block special file is then created and accessed, routines for the wrong device maybe used, or the

call may return the ENODEV ("No device") error.

Determining Major and Minor Numbers

When a driver is installed and a special device file created, a device then appears to the operating system as a file. A device is accessed by opening, reading, writing, and closing a special device file. A special device file contains the major and minor device numbers. The major number identifies a driver for a controller, such as a printer, disk drive, or terminal. The minor number identifies a specific device. On AT&T computers, the major and minor numbers for a special device file are referred to together as a device number.

Major numbers are assigned sequentially by either the system initialization software at boot time for hardware devices, by a program such as **drvinstall**(1M), or by administrator discretion. Minor numbers are designed by the driver developer are identify characteristics of the subdevice. No standard exists for the form of the minor number.

Major Numbers

Major numbers for hardware devices are determined as follows:

- **3B4000 and 3B15 computers** — the hexadecimal board code of the device from the equipped device table (EDT). Determining a new hardware device major number on the 3B4000 computer differs by the board's location on the system buses.
- **3B2 computer and SBC** — after installing the board in the computer, the **getmajor**(1M) command can be used to determine the major number.

After adding a device to the EDT, you can display the external major number with the following commands:

Table 3–2 Displaying External Major Numbers

Processor	Command
3B2	**getmajor**(1M)
3B4000 Master Processor	**getedt**(1M), **iau**(8) **disp edt**, or **getmajor**(1M)
3B15	**getedt**(1M), **iau**(8) **disp edt**, or **getmajor**(1M)
SBC	**getmajor**(1M)

The major number for a software device is assigned automatically by the **drvinstall** command. Specify a dash in the SOFT column of the master file, and **drvinstall** selects the next available number and inserts it in the master file.

On the 3B and Single-Board Computers, major numbers range from 0 through 127. The following table gives the major number ranges. If you must install a driver without benefit of **drvinstall**, then search the master files for prior usage before selecting a free number.

Table 3–3 Ranges for Major Numbers

Computer Type	Hardware	Software	Extended Bus Devices
3B2 300/400, 500/600,	1, 2, 4-15	16-19, 24-29, 44, 45, 58 59, 63, 64, 66	71-127
3B4000 ACP	0-29	30-62, 64-70	71-127
3B15	4-15,	0-2,	74-127
3B4000 MP	33-47	16-32 48-73	
3B4000 EADP	--	0, 3, 16, 19, 24, 28, 29, 58, 59, 63, 64, 66	72-127
SBC	0-15	48 - 127	--

Usually, the term "major number" refers to external major numbers. These are the major numbers used for the special device files. External major numbers for software devices are static and are assigned sequentially to the appropriate field in the master file by the **drvinstall**(1M) command; external major numbers for hardware drivers correspond to the board slot and are dynamically assigned by the **lboot** process as system boot time. The **getmajor**(1M) command returns the major number for the specified device. The **mknod**(1M) command is then used to create the files (or nodes) to be associated with the device.

Internal major numbers serve as an index into the `cdevsw` and `bdevsw` switch tables. These are assigned by the self-configuration process when the drivers are loaded, and may change every time a full-configuration boot is done. The system uses the MAJOR table (see below) to translate external major numbers (from the special device file) to the internal major numbers needed to access the switch tables.

One driver may control several devices; each device will have its own external major number, but all those external major numbers are mapped to one internal major number for the driver. Were this not the case, each driver would need a separate entry in the switch tables for each device under its control.

Minor Numbers

Minor numbers are determined differently for different types of devices. Typically, minor numbers are an encoding of information needed by the controller board, although the driver may also have information for it. For instance, for tape drives, the minor number indicates whether or not to rewind the device. Hardware device minor numbers must fall in the range 0 through 255; software device minor numbers must also fall in the range of 0 to 255.

The external minor number is entirely under control of the driver writer (although there are conventions enforced for some types of devices by some utilities), and usually refers to "subdevices" of the device. A tape driver, for example, may talk to a hardware controller (the device) to which several tape drives (subdevices) are attached. All the tape drives attached to one controller will have the same external major number, but each drive will have a different external minor number. For disk devices, the disk controller is assigned a major number, and individual disk partitions are the subdevices, with each disk partition having separate special device files and separate minor numbers.

Internal minor numbers are used with hardware drivers to identify the logical controller that is being addressed. Since drivers that control multiple devices (controllers) usually require a data structure for each configured device, drivers address the per-controller data structure by a logical controller number rather than the external major number, thus compacting the data structures in the kernel.

The logical controller numbers are assigned sequentially by the central controller firmware at self-configuration time. The controller with the lowest local bus address is assigned logical controller number zero, and so forth. The internal minor number is calculated by multiplying this number by the value of the #DEV field (number of devices per controller) in the master file.

The internal minor number for all software drivers is 0.

The MAJOR and MINOR Tables

The MAJOR and MINOR tables map internal major and minor numbers to the external major number. Each table is a character array of 128 entries. Figure 3-2 illustrates the MAJOR and MINOR tables and their relationship to `cdevsw` and `bdevsw`.

The switch tables will have only as many entries as required to support the drivers installed on the system, up to 128 entries.

Switch table entry points are activated by system calls that reference a special device file, which supplies the external major number and instructions on whether to use `bdevsw` or `cdevsw`. By mapping the external major number to the corresponding internal major number in the MAJOR table, the system knows which driver routine to activate.

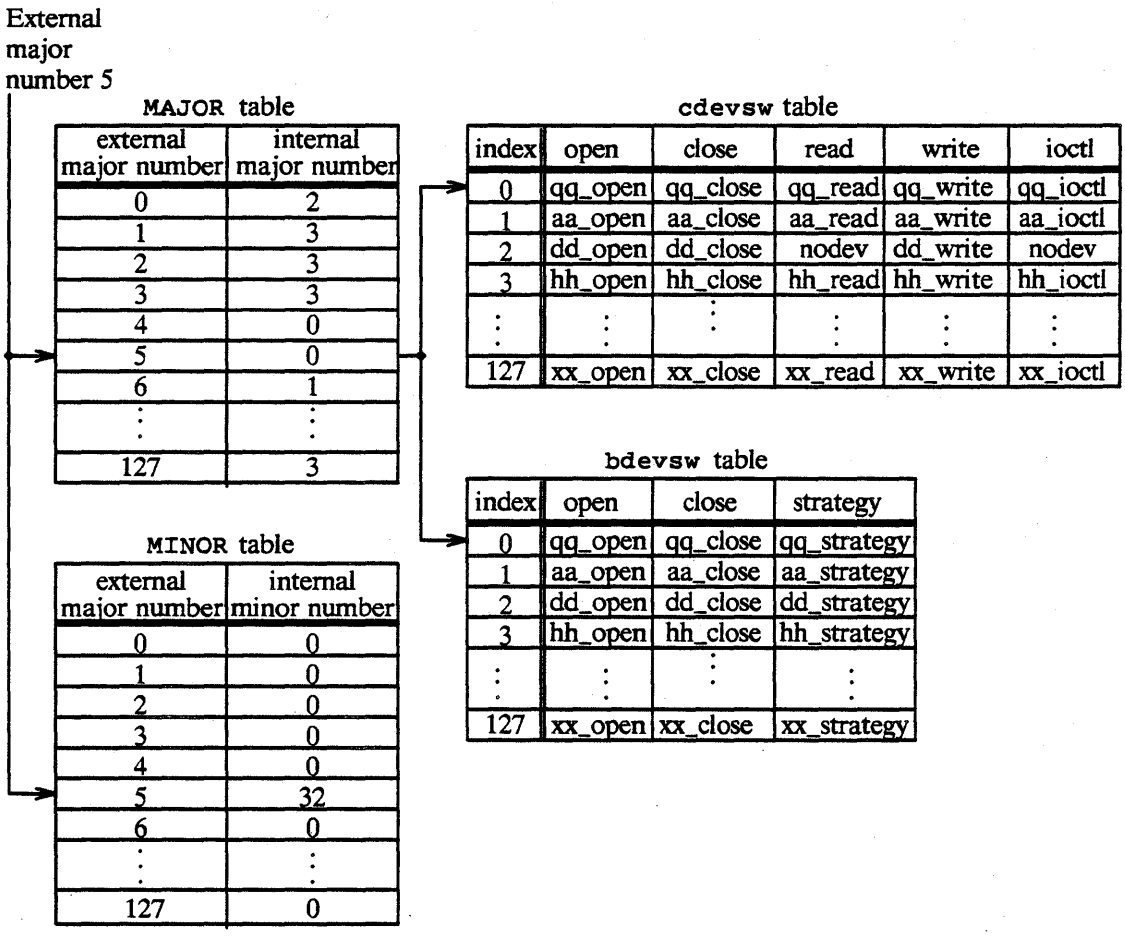


Figure 3–2 MAJOR and MINOR Tables

NOTE: In Figure 3-2, the entry "32" under the column entitled, "internal minor number" identifies that the number of total number of devices for the driver. This value is set in the master file under the #DEV column. This number is arbitrary in this circumstance.

External to Internal Translation

Driver writers usually deal directly with external major and minor numbers, and the operating system translates these to internal major and minor numbers. A driver can access an internal major minor number as follows:

- Internal major numbers can be extracted from the **MAJOR**[ ] translation table. To access the table, use the syntax:

```
unsigned char MAJOR[external_major_number]
internal_major = MAJOR[external_major_number]
```

- Internal major numbers can be determined with the built-in function `#M` in the master file, which is used to refer to the internal major number for the current driver (for example, `imaj = #M`). To refer to the internal major number of another driver in the master file, use `#M` with the name of that driver (as found in the */boot* directory) as an argument. For example `#M(MEM)`.

- Internal major and minor numbers can be accessed with the **major**(D3X) and **minor**(D3X) macros (defined in */usr/include/sys/sysmacros.h*).

Drivers should not perform external-to-internal device number translation under the following circumstances:

- During unbuffered read or write operations to "raw" devices. This translation is done when the **physio**(D3X) function calls the **strategy**(D2X) routine, as discussed in Chapter 6.

- In the **print**(D2X) routine used to handle errors arising during the execution of the **strategy** routine.

Interrupt Entry Points

The operating system handles all system interrupts, including clock and software interrupts, system exceptions such as page faults, and interrupts from peripheral devices controlled by drivers. Peripheral devices generate interrupts when an I/O transfer encounters an error or completes successfully. They also sometimes generate "stray" interrupts, which can cause general system havoc if not handled by the **logstray**(D3X) function.

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

Each device can have up to sixteen interrupt vectors assigned to it. The number of the first interrupt

vector for a device is (16 * *external-major-number*). The number of interrupt vectors for a device is determined by the value of the #VEC column in the driver's master file. So, if a driver has #VEC=4, and the external major number of the device is three, the device has interrupt vectors 48, 49, 50, and 51. See Chapter 10 for a more detailed discussion of how interrupt vectors are assigned to devices.

Each interrupt vector for a hardware device has its own driver interrupt handler, assuming the driver code includes an interrupt handler. The name of a driver interrupt handler must be either **int**(D2X), or one of **rint**(D2X) or **xint**(D2X). As with all other driver entry point routines, the driver prefix must be added to the name.

Chapter 4: Header Files and Data Structures

Contents

Introduction

This chapter describes the use of system and driver-specific header files, and the relationship between data structures and drivers. It introduces some standard system header files delivered with the UNIX operating system that define error code, parameter, and data structure information for all drivers, and describes the standard system data structure fields frequently accessed by driver routines.

This chapter also provides procedures for declaring data structures in driver code, creating driver header files for driver-specific data structures, and for defining driver-specific data structures in a driver's master file.

This chapter discusses the following:

- Using system header files including detailed information on the *errno.h* and *types.h* header files.
- Using standard system data structures including detailed information on structures defined in *user.h*, *proc.h*, *buf.h*, and *iobuf.h*. If you are already familiar with standard UNIX data structures, skip this section and turn to "Declaring Data Structures".
- Creating driver header files for defining driver-specific data structures and variables
- Defining system and driver-specific data structures in driver code
- Using the master file to define driver-specific data structures

All of the data structures introduced in this chapter are discussed elsewhere in this document. A complete listing and description of all standard system data structures currently supported for driver interface is provided in the *BCI Driver Reference Manual* in section D4X. Appendix C in this book provides a listing of common and processor-specific header files.

Header Files

A header file is a method of localizing common driver information in a file sharable by all drivers. Localizing common information reduces the overhead to the driver code itself and enhances the portability of each driver. There are two kinds of header files associated with drivers: system header files, and driver-specific header files.

The system header files included in the */usr/include/sys* directory when the UNIX operating system is delivered define a variety of standard system variables, data types, and data structures used by many or all drivers. Driver-specific header files define variables and data structures used only by the driver routines.

Each driver that uses the information contained in a header file must include the header file name at the beginning of the driver code with an **#include** line. Header files containing variable and error code information must be included in almost all drivers. The following is a listing of header files typically used by all drivers:

Table 4–1 Header Files Used by All Drivers

Header File	Description
types.h	Contains data type definitions that are required by standard system data structures; **#include** before any other header files.
param.h	Contains parameter and macro definitions required by other header files; **#include** after *types.h* in all drivers.
errno.h	Contains standard error code definitions for all drivers.
cmn_err.h	Contains the **cmn_err**(D3X) print interface definition.

Header files are called in the order they are listed. Header files that are dependent upon information contained in other header files must be included after them. For instance, the *dir.h* header file must be included before *user.h*. The *types.h* and *param.h* header files are always included before any other header files.

The following sections discuss the information contained in the *errno.h* and *types.h* header files in more detail.

Error Codes in errno.h

The *errno.h* header file defines the error codes that should be returned by a driver routine when an error is encountered. Table 4-2 lists the error values in alphabetic order. In a driver **open**(D2X), **close**(D2X), **ioctl**(D2X), **read**(D2X), and **write**(D2X) routines, errors are passed back to the user by setting the **u.u_error** field of the process user block to the appropriate error code. In the driver **strategy**(D2X) routine, errors are passed back to the user by setting the **b_error** member of the `buf`(D4X) structure to the error codes.

Table 4–2 Driver Error Codes

Error Value	Error Description	Use in these Driver Routines (D2X)
EAGAIN	kernel resources, such as memory, are not available at this time; cannot open device (device may be busy, or the system resource is not available).	**open, ioctl, read, write, strategy**
EFAULT	an invalid address has been passed as an argument; bad memory addressing error	**open, close, ioctl, read, write, strategy**
EINTR	when a process is sleeping above PZERO without PCATCH ORed to the sleep priority and a signal is received, **longjmp**(D3X) is called, control returns to user and EINTR is set in **u.u_error**.	**open, close, ioctl, read, write, strategy**
EINVAL	invalid argument passed to routine	**open, ioctl, read, write, strategy**
EIO	a device error occurred; a problem is detected in a device status register (the I/O request was valid, but an error occurred on the device)	**open, close, ioctl, read, write, strategy**
ENXIO	an attempt was made to access a device or subdevice that does not exist (one that is not configured); an attempt to perform an invalid I/O operation; an incorrect minor number was specified	**open, close, ioctl, read, write, strategy**

Table 4–2 Driver Error Codes

Error Value	Error Description	Use in these Driver Routines (D2X)
EPERM	a process attempting an operation did not have required super-user permission.	**open, ioctl**
EROFS	an attempt was made to write to, or to open a read-only device	**open**

Figure 4-1 cross references error values to the driver routines from which the error values can be returned.

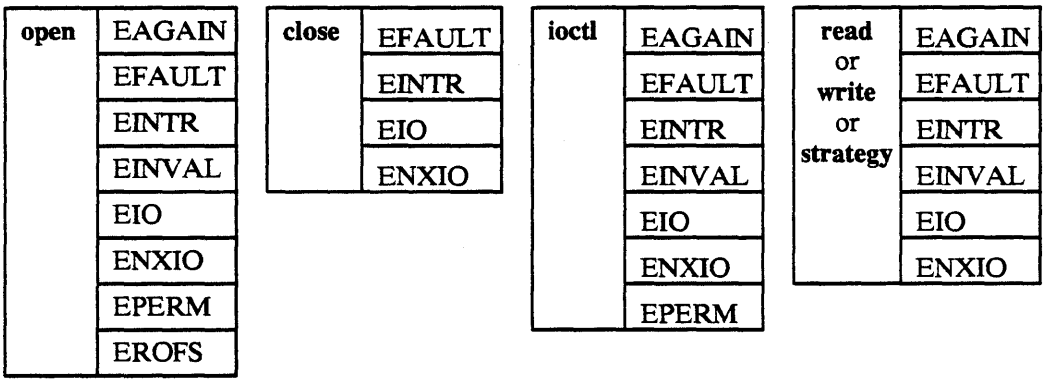


Figure 4–1 Error Codes by Driver Routine

Data Types in types.h

The header file *types.h* defines a number of special data types used widely within the kernel. Many fields in the system data structures use these types. The data type for each structure field is defined in the data structure's header file. Section D4X in the *BCI Driver Reference Manual* lists the fields in each data structure together with their defined data type.

Maintaining a standard definition for data types enhances the portability of kernel and driver code. Drivers storing values in system data structure fields must either declare variables of these types or cast the value using the C `cast` construct.

The following is a list of some of the more common data types defined in *types.h* frequently used by driver code:

Table 4–3 Common Data Types

Data Type	Description
caddr_t	virtual memory address, byte aligned
daddr_t	block device block number
dev_t	major/minor device number
label_t	setjmp data block
off_t	byte offset in file
paddr_t	physical memory address

The *types.h* and *param.h* header files should always be the first header files included in the driver code.

Drivers and Data Structures

Data structures provide a means for passing information between the kernel and the driver routines. They are used to store process status information, to define I/O transfer methods, to define buffering schemes, and to store driver and device specific information. There are basically three types of data structures: system data structures declared globally[1] for a driver, driver specific data structures declared globally for a driver, and internal data structures defined within a driver routine and used only by that routine.

System data structures are structures that define common methods of passing information to and from the kernel and device drivers. Header files for these data structures are supplied with the delivered operating system in the */usr/include/sys* directory. Driver specific data structures are structures that store information for use only by that driver and whose header files must be created by the driver writer. Internal data structures are defined within a particular driver routine and store information of use only to that routine, and often about a specific device.

Drivers declare the use of system data structures by adding the header file names with **#include** lines to the beginning of the driver code. Driver-specific data structures are declared either by their own header file or by an **extern** declaration at the beginning of the driver code. Internal data structures are not declared, but are simply created within a particular routine for the use of that routine alone.

The following sections discuss some standard data structures, provide procedures for declaring data structures, and provide procedures for creating header files for driver-specific data structures.

1. The term "global" means that the data structure has been declared at the beginning of the driver code with a **#include** line, or with an **extern** declaration.

Standard System Data Structures

System data structures are standard structures the UNIX operating uses to pass information to and from the kernel and driver routines. The header files defining these structures are delivered with the operating system in the */usr/include/sys* directory.

Many standard system data structures are used by all the computers discussed in this book (Appendix C in this book contains a more complete listing of common header files). The following header files define some of the data structures commonly used by device drivers:

Table 4–4 Common Driver Header Files

Header File	Description
buf.h	Defines the `buf` structure used for block I/O transfers.
dir.h	Defines the structure of a file system directory entry.
elog.h	Defines the `iostat` structure.
file.h	Defines UNIX file structure including flags passed to **open**(D2X) and **close**(D2X) routines.
iobuf.h	Defines the `iobuf` structure (block I/O requests) for use primarily with IDFC disk devices.
proc.h	Defines the `proc` structure used for every active process include in the process table.
tty.h	Defines the `clist` structure and commands and flags for the line discipline for TTY devices
user.h	Defines the `user` structure for the current process and is referenced by the global variable **u**.

The `user`(D4X), `proc`(D4X), `buf`(D4X), and `iobuf`(D4X), structures, always accessed when doing character or block I/O, are discussed in more detail in the following sections.

The user Structure

The `user`(D4X) structure[2] declared in the *user.h* header file defines the fields included in the user block for each process. User blocks are created dynamically for each newly created process. The process user block contains information such as where the data is coming from, its size, and how much needs to be moved. Character driver **read**(D2X) or **write**(D2X) routines may use these fields to read information they need about the status of an I/O request, and to write the I/O request's final status.

When a process begins to execute in the CPU, the process's user block is placed at a fixed address in the kernel. Only one user process can run in the CPU at one time. This means that the user block in the CPU is always the block for the current running process. A new process that has a higher priority than the process currently running may cause that process to be swapped out, in which case a new user block is swapped in for the higher priority process. For this reason, **strategy**(D2X) and interrupt routines must not access the `user` structure. These routines operate independently of the currently running process, and may alter the fields of a user block for a process not associated with them.

The majority of the fields defined in the *user.h* header file are pertinent only to character driver I/O **read** and **write** routines. **init**(D2X), **open**, **close**, and **ioctl**(D2X) routines can also access the `user` structure, however, the **u_base** and **u_count** fields that define the size and location of the data transfer are not meaningful to these routines. Block I/O requests are handled through the system buffer cache defined by the `buf` structure. See "The buf Structure" section in this chapter for information.

2. The user structure is also commonly called the **u** structure or **u** block, and sometimes referred to as the *user area*. The term *user area* should not be confused with the term *user space* which refers to the part of a system in which user processes execute.

The following `user` structure fields are of particular interest to driver routines. A † sign preceding the field name indicates the field is read-only:

Table 4–5 Fields in the user Structure

Field	Description
u_base	Contains a pointer to the virtual address of the next user data byte. The driver should increment the pointer for each byte moved. The **physio**(D3X) function automatically increments this pointer.
u_count	Contains the count of total bytes remaining in virtual address space. The driver should decrement this count each time a byte is moved. The **physio**(D3X) function automatically decrements this count.
u_offset	Contains the position in the file when the **read** or **write** was requested.
u_error	Contains the error status code for an I/O operation as defined in the *errno.h* header file. This value will be copied to the global variable **errno**, and a failure will be indicated in the system call return value if the operation was unsuccessful.
† **u_procp**	Contains a pointer to the `proc`(D4X) structure entry in the process table. The `proc` structure defines information such as the process's priority (See "The proc Structure" section in this chapter for information).

Information in the process user block is cross-referenced with information in the `proc` structure for the process. The **u_proc** field in the `user` structure contains a pointer to the process's proc structure entry in the process table. The `proc` structure defines static information such as the the process's priority level (see "The proc Structure" section in this chapter for more information).

The `user` structure is referenced by the global variable **u**. Driver code accesses `user` structure fields through that name, for example: **u.u_base**. This name refers to the **u_base** field in the `user` structure.

The proc Structure

The `proc`(D4X) structure contains information used by memory management hardware and software to locate the code, data, and stack information of the process. It also contains information used by the scheduler in selecting processes to run.

One `proc` structure is created for every process, regardless of whether it is the currently active process. In most UNIX systems, each structure is an entry to an array called the *process table* which includes all active processes and determines the maximum number of processes on a system at any time.

The process table can be accessed through the `user` structure. The **u_proc** field in the `user` structure contains a pointer to the process's process table entry. Fields in the `proc` structure can be accessed by driver routines, however, driver routines must never alter the `proc` structure fields.

The following fields in the `proc` structure are of interest to device drivers. All fields in the `proc` structure are read-only:

Table 4–6 Fields in the proc Structure

Field	Description
p_stat	Contains the status of the process and is used by the scheduler to determine the current state of the process. The process state is changed by driver calls to the **sleep**(D3X) or **wakeup**(D3X) kernel functions.
p_pri	Contains the priority of the process and is used by the scheduler to determine which process has priority for CPU use. Process priority can be changed by driver calls to the **sleep**) and **wakeup** kernel functions.
p_pgrp	Contains the process group ID of the process and is used by a driver to send signals to a group of processes.
p_pid	Contains the process ID and is used by a driver to send a signal to a specific process.
p_size	Size of the process swappable image in pages.

The buf Structure

The `buf`(D4X) structure declared in the *buf.h* header file defines the fields contained in the header for each buffer in the system buffer cache. Fields in the `buf` structure define a requested block I/O operation by specifying the device to be used by its device number, the direction of the data transfer, its size, the memory and device addresses, and other information. The kernel uses the information in the buffer header to organize and maintain the system buffer cache. A block driver **strategy**(D2X) routine uses the information in the buffer header to maintain an internal queue of I/O requests to be processed, and to store information such as the address of an I/O completion routine. Block driver **strategy** routines receive one argument, *bp*, that is a pointer to a buffer header.

The following is a list of some of the fields in the buffer header used by driver **strategy** routines. A † sign preceding the field name indicates the field is read-only.

Table 4–7 Fields in the buf Structure

Field	Description
† **b_dev**	Contains the device number (major and minor numbers) for the block device storing the buffered data.
† **b_addr**	Contains the virtual address of the data buffer.
† **b_bcount**	Contains the amount of data to be transferred in bytes.
† **b_blkno**	Contains the device number for the block device.
† **av_forw**	Contains a forward pointer for an internal queue of requests to be processed by the **strategy** routine.
† **av_back**	Contains a backward pointer for an internal queue of requests to be processed by the **strategy** routine.
b_flags	Contains information on how the I/O request is to be handled and its current status.

Driver code uses pointers to refer to fields within the buffer header. For example, the following line uses the name *bp* as a pointer to the **av_forw** field in the buffer header:

```
bp->av_forw
```

Chapter 6 in this book describes the system buffer cache and discusses a **strategy** routine's use of the fields define in the `buf` structure in detail.

The iobuf Structure

Most block device driver **strategy**D2X) routines require an internal queue to manage the device's outstanding I/O requests, since the speed with which a typical block device can service requests is considerably slower than the speed with which requests can be made.[3] **strategy** routines also need a structure to store specific device state information. The `iobuf`(D4X) data structure defined in *iobuf.h* provides fields to serve these functions.

The `iobuf` structure stores such information as the device number, an error count, the device's local bus address, and other device specific information, and provides pointers to the **av_forw** and **av_back** fields of the `buf` structure. These pointers can be used to create an internal request queue.

The following list is an example of the kinds of fields included in the `iobuf` structure:

Table 4–8 Fields in the iobuf Structure

Field	Description
b_actf	Contains a pointer to the **av_forw** field in the `buf` structure and can be used to indicate the beginning of an outstanding job request queue in the driver **strategy** routine.
b_actl	Contains a pointer to the **av_back** field in the `buf` structure and can be used to indicate the end of an outstanding job request queue in the driver **strategy** routine.
b_dev	Contains the device number (major and minor numbers) of the device.

strategy routines that wish to use the `iobuf` structure must declare the structure using the **extern** declaration in the driver's header file. The structure is a standard name constructed from the driver prefix in the form: *prefix***tab**. For example, the `iobuf` structure for the **doc_** driver included in Appendix E is declared in line 175:

```
extern struct iobuf doc_tab[];
```

3. An exception to this would be a **strategy** routine for a RAM driver. Because the data to be read or written is already in memory, requests can be serviced synchronously.

Although some form of structure is needed to provide a private I/O queue, it is not necessary to use the structure defined in *iobuf.h*. In some cases, the fields provided may not be enough to hold all the device specific information needed for your device. However, most of the fields provided are required by any structure holding device specific information, and fields from the `iobuf` structure are used in some example **strategy** routine code included in this book. For this reason, it is helpful to know the above information.

Declaring Data Structures

All system and driver-specific data structures used by a driver are declared at the beginning of the driver code. In most drivers, this is done in three steps:

1. Use an **#include** statement to reference the appropriate system header files from the */usr/include/sys* directory for system-wide data structures used in a driver.
2. Use an **#include** statement to reference the header files created for this driver and modules for such items as buffering schemes that the driver uses.
3. Declare any structures that are defined in the master file (initialized data structures). Be sure that the declaration matches the data element size used in the master file. See the "Using the Master File for Data Structures" section for information on defining structures in the master file. See the "Mismatched Data Element Sizes" section in Chapter 13 for information on checking data element sizes.

System header files should be included (using a **#include** statement) before driver-specific declarations and header files. Note, however, that it may be necessary to use a **#define** statement before some **#include** statements, for instance,

```
#define INKERNEL
```

This line should precede the following line unless the code will be compiled with the -DINKERNEL option.

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

A header file that is dependent on another header file should be included after that file. After including the system header file, include the data structures that are necessary for the new driver.

Some hardware drivers may have more than one header file. One may have the driver define instructions themselves that are used for **ioctl** calls and the interface between the driver and the user-level programs, and another may have definitions for the interface between the driver and the firmware/hardware. This latter header defines how to do operations on the board and is used by a firmware developer. For instance, a tape driver on the 3B15 computer has two header files: *tape_drv.h*, which defines data structures used when the driver interacts with the operating system, and *tape_fw.h*, which defines the firmware data structures.

Creating A Driver Header File

By creating a header file defining structures and variables specific to your driver, you make the driver easier to read and maintain. You should create your driver header file using the following conventions:

- the name must end with the "**.h**" suffix
- the name should relate to the driver, using either the name of the driver or the driver prefix
- the header file should be located in the *sys* directory that is associated with the the driver source code directory, either */usr/add-on/sys* or */usr/src/uts/sys* as well as the */usr/include/sys* directory. Note that the */usr/include/sys* directory on the 3B4000 computer has subdirectories for the Adjunct Communications Processor (ACP); *acp/sys*, Adjunct Data Processor (ADP); *adp/sys* and the Enhanced Adjunct Data Processor (EADP) *eadp/sys*, as well as a subdirectory for header files that are common to all adjuncts; *sys/adj*. Header files for drivers that run on one of these adjunct processing elements should be placed under the appropriate subdirectory.
- header files should be commented. When defining a structure, include comments that tell how each element is updated and when it is used. When defining I/O control commands in a header file (see Chapter 8), explain the use of each command thoroughly.

Because drivers are a separate part of the system, driver programmers should not change or add to standard system header files. Changing system data structures could cause user-level programs to work incorrectly if they rely on the system data structure. For example, changes to the process table will cause the **ps**(1) command to fail. In addition, modifying standard system header files makes them incompatible with standard AT&T UNIX System V.

Defining Driver-Specific Data Structures

When creating new header files and defining data structures in the driver code, adhere to the following rules:

- One **#include** file may be nested inside another. If a header file has dependencies on another header file, nested include statements ensure that the dependencies are always honored.
- The names of driver data structures and variables should use the driver name as a prefix to ease program readability and debugging, and to avoid conflict with other variables on the system with the same name.

- All declarations of structures that are allocated in the master file must be of the form **extern**.
- Static data structures can be defined in the header file or the driver code itself, but will require special initialization code. For instance

```
static int gzanyopen = 0
```

  is not valid, since the value of **gzanyopen** at boot time is determined by the value it had when the **mkunix**(1M) was run. The proper initialization code would be

```
static int gzanyopen;
...
gzstart()     {
              gzanyopen = 0;
              }
```

- Most drivers should declare a data structure for each hardware unit (device or subdevice) that may be driven by the driver. This data structure should contain a flag field to record the device status, such as "open," "sleeping waiting for data to drain," and so forth (the `iobuf` structure is a template for this kind of data structure). The majority of the contents of this data structure are device dependent so no recommendation can be given here. However, there should be one flag entry per unit, defined in the driver file and declared in the header file. If it is not appropriate to hard-code this value, it can be defined in the driver's master file and the system will calculate it at boot time; this is discussed in the next section.
- The definition of the data structures (the place in the source code where the compiler allocates memory storage) should be in the master file, especially if they are configuration-dependent. Alternatively, they can be defined in a *.c* file, usually the driver source file or its associated header file.
- Provide meaningful comments for all declarations, especially when values are set or flags for **ioctl**(D2X) routines are defined.

Defining Driver-Specific Data Structures in the Master File

The value of global data variables can be defined in the DEPENDENCIES/VARIABLES column of the master file for your driver, and then declared as a data structure in your driver. The boot software will calculate the values of the variables, allocate, and initialize the data structures defined in the master file (see Chapter 5). This practice should be used for values that might vary among machines, configurations, or usage levels (such as the size of buffers). The **master**(4) reference page and Chapter 12 list the valid operands for expressions that can be used and give instructions for creating tunable parameters in the master file.

Static variables, pointer declarations, and local structures cannot be defined in the master file but must be defined in the driver code itself. For example, the hypothetical "GZNORP" driver uses local driver data areas to buffer data begin transferred between user address space and the device (see Chapter 6 for a discussion of this I/O scheme). It uses the master file to allocate system memory for driver data areas as a function of the hardware configuration. The three elements defined are

`gzn_cnt`
: gets the number of controllers (`#C`) that the bootstrap software finds configured in the system, expressed as an integer (`%i`). For hardware drivers, this is determined by the number of boards configured; for software drivers, it is determined by a number specified on the INCLUDE line in the system file. For example

```
INCLUDE: GZNORPL(5)
```

will result in a #C=5.

`slpbuf`
: calculates the maximum number of subdevices that could be configured for this driver on the system. This is done by multiplying the number of controllers present (`#C`) by the maximum number of subdevices each controller might have (`#D`) as defined in the #DEV field. A 0x30 byte entry is allocated for each subdevice.

`gznctlr`
: allocates 0x50 bytes for each controller

The master file that defines these elements is:

```
* GZNORP
*
*FLAG    #VEC    PREFIX    SOFT    #DEV    IPL   DEPENDENCIES/VARIABLES
bca      1       gzn       -       4       6
                                                 gzn_cnt(%i) ={#C}
                                                 slpbuf[#C*#D] (%0x30)
                                                 gznctlr[#C] (%0x50)
```

Figure 4–2 Sample master File

The header file for this driver then references these variables as shown below:

```
/*   Number of gznorp controllers */
extern int gzn_cnt;
/*
 *   Bookkeeping for the devices */
extern struct gznent gznctlr[];
/*
 *   Base address for each controller's memory  */
extern paddr_t gzn_addr[];
```

In this case, the system calculates the amount of memory needed for the configuration found by the bootstrap software. If the values should be set by the administrator, you can create a tunable parameter table in the master file. Instructions for this are in Chapter 12.

The `paddr_t gzn_addr[ ]` array is the array created because of the "a" flag in the master file. For any driver with an "a" flag, **lboot** creates and fills an array named *prefix*_**addr**. This variable must not be declared as a variable in the master file, but should be declared in the driver code.

Chapter 5: System and Driver Initialization

Contents

Introduction

Device drivers must be installed as part of the kernel, and so must conform to a number of predefined specifications and procedures. For example, the driver must be declared to be of a certain type (block or character), driver routines must follow naming convention, and files must be stored in particular directories. Although the details vary from system to system, the processing required to prepare a driver for use occurs in three basic steps

- **Installation.** System files relating to the driver must be created or updated, and the compiled driver code must be installed. Instructions for completing this step are given in Chapter 12.

- **Configuration.** A new version of the kernel must be created to include information about the driver. Information must be loaded into system tables, driver structures must be created, the driver code must be linked into the kernel, and other functions must be performed. The first part of this chapter described the main steps in this process.

- **Initialization.** The newly configured kernel is then executed. System processes are begun, and the driver initialization routine (either **init**(D2X) or **start**(D2X)) is executed. At the end of this chapter, example driver initialization routines are presented along with guidelines for determining what initialization may be needed for different types of drivers.

Many of the details of system configuration and initialization are independent of driver initialization. They are included in this chapter mainly to help debug the driver. Errors in the driver **init** or **start** routine may cause a system crash soon after booting. In that case, it is very helpful to have a clear idea of what happens when the system is booted.

System Configuration

The next few sections cover some of what driver developers should know about system configuration.

Driver Files Needed for Self-Configuration

Before booting the system and invoking self-configuration, the following files must be created or updated. These are discussed more fully in Chapter 12 and Appendix A.

- **master file** — provides driver-specific information, such as whether it uses the block or character interface, the interrupt priority level (IPL) for the device, and dependencies this driver has. Self-configuration does not itself access the master file; rather, the master file information is incorporated into the bootable executable file in the */boot* directory.
- **bootable executable file** — the driver object code, residing in the source code directory, with the information from the master file built into the *optional header* section (see */usr/include/a.out.h*). The **mkboot**(1M) command creates this file in the */boot* directory.
- **Equipped Device Table (EDT)** — a table that lists all hardware devices present on the system, taken from the */dgn/edt_data* file.
- **system file** — identifies software drivers that should be included and hardware drivers which, though present, should not be included in this kernel.

The files in */boot* have upper case names; the corresponding files in */etc/master.d* have lower case names.

Starting Self-configuration

Installed drivers are configured into the operating system kernel when the system is booted. The system firmware provides the pre-bootstrap processing, including running diagnostics, initializing mainstore, building the EDT, and starting UNIX kernel booting by calling **mboot**. **mboot** calls **lboot**,[1] which builds the kernel, including the drivers.

The **mboot-(olboot)-lboot** sequence is called self-configuration. Once the driver is installed, self-configuration makes it a functioning part of the operating system kernel.

1. On some systems, **mboot** calls **olboot**, which in turn calls **lboot**.

Self-configuration has two modes of operation. The mode in which self-configuration runs is determined by the type of file self-configuration is told to load. On the 3B15 computer and the 3B4000 master processor, the operator tells the self-configuration process which file to load by responding to the `Enter path name:` prompt that appears after the **boot**(8) command is issued. On the 3B2 computer, the computer's firmware automatically displays an `Enter name of program to execute [ ]:` prompt.

The first mode, which runs when the name of a system file is provided, is referred to as the *autoconfig* or full configuration boot. In this mode, the hardware and the system configuration file are examined to determine what drivers are to be configured into the kernel. The second mode is referred to as the absolute boot mode, or more commonly, "boot of */unix*". In this mode, a boot image is loaded. Most routine booting of the system is done in the absolute boot mode.

When the self-configuration process is complete, system initialization begins.

On the 3B4000 computer, the adjuncts are booted only after system initialization is completed for the Master Processor. The adjuncts go through a self-configuration and system initialization process similar to that of the Master Processor. On the ACP, self-configuration runs on the ACP with the ACP integral disk housing boot critical files. Self-configuration for the ADP and EADP is controlled by user-level processes that run on the Master Processor.

Steps in Self-Configuration

In effect, the self-configuration process acts as a dynamic link editor. It performs the following functions of interest to driver developers:

- creates the driver structure list
- downloads pumpcode to the pumpable device (3B15 and 3B4000 MP only)
- checks symbolic values
- assigns internal major numbers
- generates system tables
- generates interrupt vectors
- loads driver structures

- copies driver code and */boot/kernel* code into RAM and link-edits
- begins the system initialization process by passing control to the kernel physical startup routines

The most important of these steps are described below.

Creating the Driver Structure List

The driver structure list is an internal linked list created by the self-configuration process. It contains one structure for every driver that has an entry in the */boot* directory. At the head of the list is the kernel data structure, which is similar to the driver structure except it has fewer fields. Each entry is marked either INCLUDE or EXCLUDE based on whether there are any corresponding devices in the EDT and entries in the */etc/system* file.

If an included driver is dependent on an excluded driver, (as indicated in the master file) neither driver will be configured into the operating system. Error messages will indicate that the driver was excluded.

Figure 5-2 illustrates the structure of each driver in the list. The number of controllers is determined by:

- the EDT (for hardware drivers)
- the INCLUDE line in */etc/system* file (for software drivers)
- for required drivers ("r" under FLAGS in master file), the value is always 1

All drivers must have a *.text* section. If the driver object code does not include a *.bss* or *.data* section, **lboot** creates a dummy header for a zero-length section.

`struct driver *next` pointer to next driver in the list
`char *name` driver name (corresponds to */{path}/boot* name)
`struct master *opthdr` optional header from driver object file (contains master file information)
`unsigned char flag` flags (from */{path}etc/master.d*)
`unsigned char nctl` number of controllers (expansion of #C variable in master file)
`ushort int_major` internal major number
`unsigned char ntc_lu` number of logical units across HA (used for SCSI devices only) (expansion of #S variable in master file)
`unsigned char maj[MAXCNTL]` external major number of each controller
`unsigned char sys_bits[MAXCNTL]` corresponding ELB sys-bits for devices on 3B15 LBE
`long timestamp` (f_timdat from file header)
`long nsyms` number of symbols (from filehdr in object file)
`long symptr` pointer to symbol table
.text section header (from driver object file)
.data section header (from driver object file)
.bss section header (from driver object file)

Figure 5–1 Driver Structure

Downloading Pumpcode (3B15 computer and 3B4000 MP only)

Pumpcode can be downloaded to a device by the driver's **start**(D2X) routine, which executes after the self-configuration process is completed. It can also be downloaded by an **ioctl**(D2X) routine or by a script in the */etc/rc.d* directory. However, the 3B15 and 3B4000 MP support downloading pumpcode to a device requesting it, so **lboot** must handle it. This is typically used for boot devices. The downloaded code is never used during self-configuration.

When the boot process begins, it accesses the bootstrap programs from the unpumped boot device. This implies that the firmware of the boot device does not rely on pumpcode for all its software. After the driver list is populated, **lboot** creates structures in kernel address space, then loads pumpcode from the */lib/bootpump.d* directory[2] into these structures. The pumpcode structures are then matched to the corresponding driver structures, and the pumpcode is downloaded to the appropriate device.

After the configuration table is printed and the kernel and all drivers are loaded, **lboot** instructs the controllers to start executing the downloaded code. This is the last thing done before calling the UNIX system to start initializing.

Checking Symbolic Values

Before creating the symbol table, **lboot** checks that no symbolic name has been defined more than once. All symbolic names declared in the master files as well as those declared as **extern** in the driver code are compared, including those for drivers that are excluded. If **lboot** finds a name with more than one value, it first attempts to resolve it by checking that none of the multiple values are defined for excluded drivers. If so, it prints a warning message and proceeds. If there are multiply-defined symbols for non-excluded drivers, **lboot** initializes them to zero. While this allows **lboot** to continue, it may cause the system to panic or seriously malfunction before the boot process completes.

lboot also looks for referenced but undefined symbols. If it finds an undefined symbol, an error message is printed and the symbol is initialized to 0. This condition may also cause the system to panic or seriously malfunction.

Generating System Tables

The MAJOR and MINOR tables are character arrays of 128 entries. For each external major number, **lboot** inserts the corresponding internal major number it has calculated into the appropriate slot in the MAJOR table. Only one internal major number is assigned to each driver, whereas each device controlled by a driver has its own major number. Consequently, several internal major numbers (several devices) may map to the same internal major number (same driver).

lboot determines the external major numbers in one of the following ways:

- External major numbers for software drivers are listed under the SOFT column of the master file; **lboot** gets this information from the optional header member of the driver structure list.
- External major numbers for most hardware devices correspond directly to the slot in which they are installed, and **lboot** uses these numbers.
- The 3B15 computer supports an extended local bus unit (ELBU); major numbers for devices on the ELBU are *32 + board address*. The **lboot** process calculates the major numbers for ELB devices, then writes these values to the MAJOR table.

The type of access supported by a driver is determined by a "b" or "c" in the FLAGS column of the master file. **lboot** gets this information from the **flag** member of the driver structure.

This two-pass approach is taken to limit the size of the `bdevsw`(D4X) and `cdevsw`(D4X) tables.

At this point, **lboot** generates the `bdevsw` and `cdevsw` tables and the corresponding **bdevcnt** (number of block-access devices) and **cdevcnt** (number of character-access devices) values.

Generating Interrupt Vectors

lboot determines the number of required interrupt vectors by adding the numbers from the #VEC column of all master files. It then sets up a single interrupt vector table, which is used to access the drivers' interrupt routines.

Regardless of what is coded in the driver, **lboot** determines whether to use **int**(D2X) or **rint**(D2X)/**xint(D2X)** pair for the interrupt routine(s) for each device according to the ratio of the number of vectors per device (#VEC) to the number of subdevices per controller (#D). If the number of vectors is double the number of devices, **lboot** will create two interrupt vectors per subdevice and expect the **rint/xint** pair of routines. Otherwise it will expect the **int** routine.

---

2. Files in this directory must be named *board-name***pump**. For instance, if the board name is **ports**, the pumpfile must be named **portspump**.

To populate the interrupt vector table, **lboot** creates an assembly assist routine that pushes the device number onto the stack, then calls the driver interrupt handler routine.[3] It then puts the address of the interrupt assist into the table and assigns the appropriate interrupt priority level (IPL) to each vector.

Each device can have up to sixteen interrupt vectors assigned to it; see Chapter 10 for an explanation of how the interrupt vector numbers correspond to the external major number of the device.

Loading Driver Structures

Before loading the driver structures, **lboot** calculates values for all driver variables and symbols and adds them to the symbol table. It first computes values for variables defined in the master files, then those defined as **extern** in the driver code, and finally **static** symbols defined in the driver code.

For **extern** symbols that are defined in the driver, **lboot** computes the final value and saves the original value.

The system is loaded in several steps.

1. First loaded are all sections of the kernel that run in physical addressing mode (those whose names do not begin with "."). Undefined symbols are relocated. These sections occupy the lower portion of mainstore.

2. Next loaded are all sections of the kernel that run in virtual addressing mode (those whose names begin with ".") except for *.text*, *.data*, and *.bss*. Special symbols are defined (S*name*, E*name*, and *name*SIZE, where *name* is the name of the section without the initial "."). The section corresponding to virtual address 0 must exist and be loaded; its real address is stored so that interrupt vectors can be inserted. Each section is loaded at the next highest word boundary.

3. Location counter for the kernel *.text* and *.data* sections are assigned.

4. The *.text*, *.data*, and *.bss* sections of the kernel object code are loaded, relocating undefined symbols. The special symbols (E*name*, S*name*, and *name*SIZE) are loaded for these sections.

5. The driver structure list is loaded.

6. Driver data structures are generated. They will be initialized by the drivers' **init** routines when the self-configuration process is complete.

---

3. If the driver code includes an interrupt handling routine of any sort, **lboot** will create either an **int** or **rint/xint** assembly assist routine in the interrupt vector table, according to the ratio of #VEC to #DEV in the master file. **lboot** will call the routine(s) that it creates; as long as the driver was coded with the same routine, there are no problems. This is discussed more in Chapter 10.

7 The **io_init** and **io_start** tables are created. These structures are used to access the **init** and **start** routines of the drivers, since these routines do not have entries in the device switch tables.

8 The real addresses for the *.bss* sections are assigned.

9 The `sys3b` symbol table is completed.

10 The 3B4000 and 3B15 computers record the pathnames of any pumpfiles that were used in a special section of the operating system.

At this point, control is passed to the physical entry point for the kernel, which begins system initialization. Effectively, **lboot** has resolved several *.o*-like files into a fully-resolved *a.out*-like file.

Driver Rules Enforced by Self-Configuration

The self-configuration process imposes coding restrictions for device drivers and configurable modules. These restrictions arise as a result of the dynamic linking of the kernel and configuration modules at boot time. These restrictions and requirements are

- Never assume that globally initialized, dynamic data is properly initialized; it must be explicitly initialized in the driver code. There can be no **static** variables whose initial contents are depended on by code fragments. Such items as "first-time" switches, lock words, and initial pointers for linked lists are not allowed. The only initial value that can be assumed is zero for variables allocated in the *.bss* section. (This restriction, however, does not apply to statically allocated and initialized identifiers used as constants.) Further, any initialized data may be different in the */unix* file that is created later.

- There can be no references to routines or identifiers defined within other modules unless there is a strict dependency chain established by the dependency list in the master file. The single exception is a reference to a routine in another module which is defined in the routine definition lines of that module's master file entry.

- Any necessary data areas must be definable using the capabilities of the variable definition lines in the master file. Furthermore, the sizes of all such data structures must be adjusted based on the configuration that exists at configuration time, using the capabilities allowed by the master file.

- Drivers must be written to expect the entire device number (composed of the major and minor numbers) passed in their argument lists rather than just the minor number. This is not true for drivers written for non-self-configuration systems. A device number must, in general, be processed in the following three steps:

    1 The minor number must be inspected to determine that it refers only to devices on an individual controller.

    2 The **minor**(D3X) macro must be invoked to convert the device number into an internal minor number.

    3 This internal minor number must be verified to ensure that it only refers to an existing device.

- Any peripheral device on the system must be under the direct control of only one driver on the system. Drivers that interface to hardware indirectly do not violate this requirement.

- Any interrupt routines required for a peripheral must interface to one and only one driver.

System Initialization Process

When the self-configuration process is completed, it begins system initialization by calling the physical entry point of the kernel. System initialization initializes the kernel and drivers, creates process 0, executes the **init**(1M) process, and starts the system processes such as the swapper.

Briefly, system initialization is executed in the following order:

1. The physical memory manager and the mapping parameters array are initialized, and the virtual-to-physical mapping information is generated. The gate, interrupt, and exception tables are the first to be mapped, followed by the kernel *.bss*, *.data*, and *.text* segments. The following two sections outline the virtual to physical memory mapping.

2. All driver **init**(D2X) routines are run. Driver **init** routines are in the init data array.

3. The root file system is mounted internally in the kernel. Note that no entry is made in the *mnttab* file at this point. The **bcheckrc** process that is run by **init** will zero out the *mnttab* file and then create an entry for root in the mount table.

4. All driver **start**(D2X) routines are run. Driver **start** routines are in the **io_start** data array.

5. After the driver **start** routines have been executed, the system processes are started, including **sched** and **init**(1M). **init** is a general process spawner, whose primary role is to create processes as specified in the */etc/inittab* file. See the "The /etc/inittab File" section in this chapter for information on the structure of *inittab* and related files and directories.

Gate and Interrupt Vector Tables

System initialization begins in physical mode. It first initializes the physical memory manager and the mapping parameters array, then generates the virtual-to-physical mapping information in low memory for the items listed below and in the next section. After completing all the mapping, the system allocates table space, then retrieves these parameters and uses them to build the appropriate Segment Descriptor Tables (SDTs) and Page Descriptor Tables (PDTs).

The following tables and vectors are mapped at fixed locations by the *gate.c* file:

First level gate table	Location: virtual address 0. Although the hardware defines 32 entries, the UNIX operating system only uses entries 0 and 1.
Process and stack exception vectors	Locations: process exception, physical address 0x84, stack exception, physical address 0x88.
Interrupt vector table	Location: physical address 0x140. The hardware defines 256 entries, each of which is defined as a kernel fixed process control block. The second entry in the interrupt vector table is the process switcher (PIR #1), and the third entry is for callout processing (PIR #2). Any entries that are not used are assigned a null process control block and logged as stray interrupts.
Second level gate table	The system call cage table that prevents unauthorized entry to a system call throughout GATE 0. Note that *os/trap.c* is responsible for checking that normal exceptions through GATE 1 are valid. On the SBC and 3B2 computers, this table has 64 entries; on the 3B4000 and 3B15 computers, it has 152 entries.
Normal exception gate table	Contains normal exception entry points defined in *ttrap.s*. This is the gate table that faults the user process that attempts invalid gate access as well as page faults and other faults. It is indexed by the internal state code field in the program status word (PSW).
Dummy gate vector	Catches user code that does a GATE with register zero set to anything other than a 0 or 1. On the SBC, 3B2, and 3B15 computers, this table has 29 entries; on the 3B4000 it has 197 entries.

Other Virtual-to-Physical Mapping

After the gate and interrupt vector tables are mapped, the remaining virtual-to-physical mapping is done in the following order:

1. kernel *.text* segment
2. kernel *.data* segment
3. kernel *.bss* segment
4. first segment of the central controller (CC) board (128K)
5. second segment of the CC board (128K)
6. scratch segments (each up to 1 page)
7. primary local bus I/O space
8. incore file system (3B4000 adjuncts only)
9. additional I/O space for extended local bus, if any
10. dynamic kernel segments
11. page frame identity map (**pfdat**), which is an array of structures containing page frame information. This structure contains an entry for every unallocated page of memory left in the system.
12. all remaining free memory

At this point, the Memory Management Unit (MMU) tables (process table pointers, proc table, and region tables) are initialized. These tables are statically allocated in the kernel master file, beginning in the first page of the free memory area mapped in **pfdat**.

The mainstore cache, console, and the second console port (contty) UART interrupt devices are also initialized. Then the kernel zeros its *.bss* space, including the drivers.

The /etc/inittab File

The */etc/inittab* file controls the processes executed by the **init**(1M) program when the computer is initialized and any time the computer changes run level. When a new state is entered, the **init** program reads *inittab*, finds the "instructions" that apply to that run state, and executes those programs in the order in which they are listed in *inittab*. For most drivers, you will not modify *inittab* but rather create other files that will be called automatically.

Each line in *inittab* has four fields, separated by colons. A comment should be added at the end of the line; it is preceded with a "#" and can go to the end of the line. The four fields are:

id One or two characters used to uniquely identify an entry.

rstate The state or states in which this command can be executed. The valid values with their meanings are:

value	state
s,S,0,1	Single-user state
2	Multi-user state
3	Multi-user state with RFS running
4	Not currently used
5	Go to firmware mode
6	Automatic reboot

NOTE: 0 in **rstate** means power down on the 3B2 compute and single-user on the 3B15 or 3B4000 computers. If no number is specified, the default is that the command can be executed in any run state.

More than one number can be used in this field; for instance, "56" means to execute this process when the system state switches to either state 5 or 6.

action The conditions under which **init** should execute the process in this line. For a full explanation of all actions, see **inittab**(4) in the *UNIX System V Programmer's Reference Manual*. The options of interest to driver writers are:

wait start process and wait for it to terminate when system first enters that runstate

bootwait execute only once after system is booted, the first time the system enters a state that matches **rstate** for this entry.

off do not restart this process when state changes

sysinit used for initializing devices, identifies entries to be executed before **init** spawns a shell on the console

respawn restart this process if it dies or if it is not already running when system state changes

process The full pathname of the process to be invoked and arguments to the process

Figure 5-2 is an example of a pristine */etc/inittab* file.

```
 1   # /etc/inittab file
 2   #
 3   fs::sysinit:/etc/bcheckrc </dev/console >/dev/console 2>&1
 4   xdc::sysinit:sh -c 'if [ -x /etc/rc.d/0xdc ] ;
           then /etc/rc.d/0xdc ; fi' >/dev/console 2>&1
 5   mt:23:bootwait:/etc/brc </dev/console >/dev/console 2>&1
 6   pt:23:bootwait:/etc/ports </dev/console >/dev/console 2>&1
 7   is:s:initdefault:
 8   p1:s1234:powerfail:/etc/led -f  # start green LED flashing
 9   p3:s1234:powerfail:uadmin 2 0
10   f1:056:wait:/etc/led -f      # start green LED flashing
11   s0:056:wait:/etc/rc0 >/dev/console 2>&1 </dev/console
12   s1:1:wait:/etc/shutdown -y -iS
           -g0 >/dev/console 2>&1 </dev/console
13   s2:23:wait:/etc/rc2 >/dev/console 2>&1 </dev/console
14   s3:3:wait:/etc/rc3 >/dev/console 2>&1 </dev/console
15   of:0:wait:/etc/uadmin 2 0 >/dev/console 2>&1 </dev/console
16   fw:5:wait:/etc/uadmin 2 2 >/dev/console 2>&1 </dev/console
17   RB:6:wait:echo "0he system is being
           restarted." >/dev/console 2>&1
18   rb:6:wait:/etc/uadmin 2 1 >/dev/console 2>&1 </dev/console
19   he:234:respawn:sh -c 'sleep 20 ;
           exec /etc/hdelogger >/dev/console 2>&1'
20   co:234:respawn:/etc/getty console console
21   ct:234:off:/etc/getty contty contty    # Network out
```

Figure 5–2 Example /etc/inittab File

Directories and Files Called by /etc/inittab

The */etc/inittab* file calls a number of programs that either execute actions or execute the files in certain system-specific programs. Whenever possible, you should add to these files and directories rather than augment */etc/inittab* itself. Any mention of shell scripts in this section can mean an executable "C" program in addition to a shell script. Table 5-1 summarizes these files and directories; the following sections describe each in more detail.

Table 5−1 Directories and Files Called by /etc/inittab

Program	rstate	action	executes:
/etc/brc	2	bootwait	files in */etc/brc.d* directory
/etc/rc2	2	wait	files in the */etc/rc2.d* directory and then the files in the */etc/rc.d* directory
/etc/rc3	3	start rfstart stop	Starts RFS Initializes variables Stops RFS
/etc/rc0	56	wait	self

/etc/brc.d

The **/etc/brc** program executes the shell scripts in the */etc/brc.d* directory, in alphabetical order. This happens once upon the first transition to multi-user state after booting, after the file systems are checked but before they are mounted and the daemons started. These scripts set up protocols and clean up the system before the file systems are mounted and daemons started. This is a good place to start a driver that is needed only when the system is in multi-user state. For instance, on the 3B15 computer, the Input/Output Accelerator (IOA) is configured at this point.

On the SBC and 3B2 computers, the **/etc/ports** command that creates special device files and entries in the */etc/inittab* file for the ports boards is run after **brc**.

/etc/rc.d The **/etc/rc2** program executes shell scripts that start with S or K in the */etc/rc2.d* directory and then executes the scripts in the */etc/rc.d* directory in alphabetical order. */etc/rc.d* is only searched for historic compatibility. New scripts should be placed in */etc/rc2.d*. The first file to execute mounts the file systems that are listed in */etc/fstab*. Most drivers should be initialized before this happens, but you may have related processes to start at this point. For instance, the **errlog** daemon associated with the **errlog** driver on the 3B15 computer and 3B4000 master processor is started here.

On the SBC and 3B2 computers only, **rc2** runs the **/etc/disks** program that recreates special device files for all "disk" subdevices in */dgn/edt_data*. You should put a file here to create the special device files for your device, unless it is an actual terminal port (not a network or printer that uses a TTY port) or a disk. Because the external major number of a device on these machines may be changed by the addition/removal of another device, special device files should be recreated every time the system is booted. On the 3B15 and 3B4000 computers, the major number of a device changes only if the board is physically moved, so this step is not necessary.

/etc/rc3.d These scripts are executed by the **/etc/rc3** program when the system goes to state 3, which is multi-user state with Remote File Sharing (RFS) running. Driver-associated processes that should run only when RFS is running should be started here.

/etc/rc0 The */etc/rc0* script controls the shutdown process. In general, processes that are started by either **brc** or **rc2** should be explicitly stopped in */etc/rc0*.

3B4000 ABUS Bootstrap Process

On the 3B4000 computer, the ABUS bootstrap process boots the adjunct processing elements after system initialization is completed for the Master Processor. This is done automatically when the system goes to multi-user state (state 2 in the *inittab* file), or can be initiated manually from the console.

The ABUS bootstrap provides functionality similar to the standard UNIX system bootstrap discussed above, but it consists of several user-level programs that execute on the master processor. The **bootape**(1M) command boots an adjunct; the **bootabus**(1M) command calls **bootape** to boot all configured adjunct processing elements.

Driver Input to the ABUS Bootstrap

The files and data required by the ABUS bootstrap process are similar to those used for the UNIX bootstrap. The */adj* directory on the Master Processor contains a subdirectory for each configured adjunct processing element. These subdirectories are named */adj/pe#* where "#" represents the processing element number (for example, "pe8" and "pe106"). The ABUS bootstrap gets its information about drivers from

- **master file** in the */adj/pe#/etc/master.d* directory
- **bootable executable file** in the */adj/pe#/root* directory
- **EDT data file**, which is */adj/pe#/edt*
- **system file**, which is */adj/pe#/etc/system*
- **special device files** for the MSBI and each adjunct are in the */dev* directory; special device files for peripheral devices on the adjuncts are in the */adj/pe#/dev* directory

Pre-Bootstrap Processing

ABUS booting begins by ensuring that the MSBI is in an operational state; if it is not, **bootabus** downloads the MSBI operational firmware[4] which allows communication to the Master Processor over the Maintenance Access Path (MAP) port. Next, the MSBI diagnostics are downloaded over the MAP port and executed.[5]

4. The firmware is downloaded by the **/etc/msbidl** command; the firmware download file is in */lib/msbi_image*.
5. The diagnostics are downloaded and executed by the **/etc/dgndl** command; the firmware download file is */lib/dgn/msbi/selftest*.

Once the MSBI is operational, **bootabus** spawns a **bootape** process for each configured adjunct processing element. All adjuncts are bootstrapped in parallel.

Booting an adjunct consists of the following:

1. verifying that all special device and configuration files For the adjunct exist and are of the correct type[6]
2. checking if adjunct is in a bootable state (not running or being booted)
3. running ROM-resident diagnostics and verifying the results
4. executing the adjunct Self-Configuration process (**/etc/unixgen**).
5. downloading the **/lib/adjboot** stand-alone process to the adjunct over the MAP port. This provides the protocol that allows the adjunct to communicate over the ABUS.
6. adding the adjunct's incore file system (*/adj/pe#/dev/icfs*) to the */etc/mnttab* file on the Master Processor
7. executing the **/etc/adjrc** command which executes the scripts in the */adj/pe#/etc/rc.d* directory

ABUS Self-Configuration

Full self-configuration for an adjunct is similar to full self-configuration for any UNIX system, with the following exceptions:

1. It creates an incore file system for the adjunct using **/etc/mkfs**(1M).
2. It does not download code to controllers.
3. For file servers, it creates the adjunct *edt* file using the SCSI **edtgen** utility that downloads a process that generates a temporary EDT called "inquiry data," then uses this inquiry data information to create the adjunct *edt* data file.
4. The EDT is a data file (named *edt*) rather than a table in ROM.
5. It builds the I/O data structures for the adjunct kernel and fills in the switch table entries. The interrupt assist routines and **pcbs** are not generated for the file server and computational server.

6. The following files are checked: */dev/pe#* and */config/pe#/pe*, where # represents the processing element number.

6 It creates the `sys3bboot` structure that contains system configuration information. The `bootpump` and `e_dumpdev` structures are not created for an adjunct.

7 **bootape** uses the **cc**(1) compiler and the **ld**(1) link editor to create the boot image (in the */adj/pe#/dev/unix* file), then downloads this boot image to the adjunct and executes it. Regular UNIX system self-configuration creates this boot image after system initialization is completed, whereas adjuncts are always booted from this image.

Adjunct Operating System Initialization

The operating system initialization of an adjunct kernel is similar to regular UNIX system initialization. It creates the virtual-to-physical mapping, zeros its *.bss* space (including drivers), and creates the environment for process 0.

The driver initialization routines are called, the kernel's I/O system and file system initialization functions are called, and the incore file system is mounted.

At this point, the system processes are started. The adjunct operating system does not have an **init** process, so the kernel idles while waiting for work.

Initializing Drivers

The tasks involved in initializing drivers differ for hardware and software drivers. Hardware driver initialization can include the following:

- clearing flags and counts previously set by the driver
- setting interrupt vectors
- allocating resources
- initializing kernel structures and pointers required for device communication
- initializing the hardware device or devices
- determining whether the device or devices are online

Software driver generally require a less complicated initialization since there is no actual device. Software driver initialization can include the following:

- initializing kernel data structures used by the driver
- allocating resources such as a memory map

A driver can be initialized by one or a combination of the following driver routines:

init(D2X)
: An **init** routine can be used for any driver that does not need access to the *root* file system in order to initialize, such as a driver that is downloading pumpcode from disk. An **init** routine must be used with drivers for devices that the kernel uses to initialize itself. A driver need by the kernel for kernel initialization is indicated by an "r" in the FLAG column of the driver's master file.

start(D2X)
: A **start** routine can be used for any driver and must be used for drivers that need access to the *root* file system in order to initialize.

ioctl(D2X)
: **ioctl** routines can be used for hardware device drivers if the device needs to be initialized in different ways for different configurations. For instance, the 3B15 computer's IOA driver is initialized with I/O control commands so that appropriate protocol-dependent scripts for the devices supported by a specific IOA can be downloaded.

open(D2X)
: An **open** routine can include initialization functions that should be run each time the device is opened.

Drivers can be initialized through a combination of the above routines at different times. For example, the **init** or **start** routine for a hardware driver could initialize any kernel data structures required for the device, but not initialize the device itself. The device initialization (such as sysgening the board and setting the board's bit configuration) might be done with the **ioctl** and **open** routines activated by user-level programs after the operating system is running.

Driver init and start Routines

Most drivers have either an **init**(D2X) or a **start**(D2X) routine, although it is quite permissible to use both for one driver. A driver must have either an **init** or **start** routine if

- the driver needs kernel structures other than the standard structures (such as `clist`(D4X)) that are part of the operating system
- the driver has static data (data that is private to that driver). Static data is put in the kernel's *.data* area. When an absolute boot is done, the initial contents of the *.data* section are the same as when the **mkunix** command was executed. If the driver modifies the static data, it must use an **init** or **start** routine to reinitialize it every time the system is booted.

The **init** or **start** routine must initialize any arrays or data structures used in the driver code, and do any set up required by the specific device such as resetting or establishing default parameters.

Example Initialization Routines

The following sections show some different initialization routines that have been written. Each driver has its own particular initialization needs, but by studying these examples you can learn the sorts of checks and error handling that is done in initialization routines and how drivers initialize structures and set up pointers and registers that are needed to communicate with a device. Initialization of TTY drivers is discussed in Chapter 7.

Initialization Routine for a Software Driver

The simplest sort of initialization routine is that of a software driver, since all that is usually required is to initialize kernel data structures that are needed for the driver. As an example, Figure 5-3 shows the **msginit** routine from the **msg** driver, which initializes the `msgmap` message allocation map. Technically, **msg** is a module not a software driver, but the principles are the same.

This initialization could also have been done with a **start**(D2X) routine. It uses **kseg**(D3X) and **btoc**(D3X) to allocate the memory, based on values set through the master file. This makes it possible to change the amount of memory being allocated without recompiling the driver. It initializes a private space management map with the **mapinit**(D3X) function, and frees all the space in the map with the **mfree**(D3X) function.

```
1    msginit()
2    {
3        register int        i;    /* loop control */
4        register struct msg *mp;     /* ptr to msg begin linked */
5        extern char msgsegment[];

7        /* Allocate physical memory for message buffer. */

9        if ((msg = (paddr_t)kseg(btoc(msginfo.msgseg * msginfo.msgssz))) ==
10           NULL) {
11           cmn_err(CE_NOTE,"Can't allocate message buffer.\n");
12           msginfo.msgseg = 0;
13       }
14       mapinit(msgmap, msginfo.msgmap);
15       mfree(msgmap, msginfo.msgseg, 1);
16       for (i = 0, mp = msgfp = msgh;++i < msginfo.msgtql;mp++)
17       mp->msg_next = mp + 1;
18   }
```

Figure 5–3 Software Driver Initialization Routine

Initialization Routines for Hardware Drivers

The **doc_** driver code given in Appendix E provides a good example of how to initialize a hardware device. This is a disk device driver that runs on the SBC computer, but is illustrative of hardware device initialization in general. The **doc_** driver is initialized through a combination of the following routines:

- **doc_init**, the initialization entry point routine, that begins at line 283.
- **doc_initdr**, a subordinate routine called by **doc_init**, that begins at line 540. It initializes drive parameters in the controller.
- **doc_open**, the entry point routine, that begins at line 592. It sets the physical description for the device the first time it is opened.

Descriptions of each routine are provided in Appendix E.

Initializing Intelligent Devices on the 3B15/3B4000 Computers

To initialize an intelligent device, you must download code and initialize the queues that associate interrupts with a particular subdevice, then *sysgen* the device. Sysgen is the procedure used to inform a controller of the location, number of entries, and size of queues that a driver will use to communicate with a controller.

The 3B15 and 3B4000 computers include the **drv_rfile**(D3X) to read a file into a buffer that it creates. This function simplifies the coding required to pump files to an intelligent controller. Since this function is not available on other machines, code that uses it should be isolated into a subordinate driver routine which the initialization routine calls only for `#if u3b15`. If the driver is ported to other machines, alternate subordinate routines can be provided that provide the functionality without using **drv_rfile**.

The **start** routine from the hypothetical **gzn** driver (Figure 5-4) is a good example of how an intelligent device is initialized on the 3B15 and 3B4000 computers.

Each controller's microprocessor is driven by code which is downloaded ("pumped") onto the controller during the boot process. This downloaded pump code is stored in a file in the form of a binary memory image which is simply copied into the RAM memory of the controller. While the download is being done, the controller executes from ROM code installed on the board. To effect the transfer of control from ROM to pump code, a forced-call command is sent to the controller. If the download attempt fails, the on-board ROM code may provide a "fall back" mode of operation with some degree of functionality.

The **gznstart** routine does the following:

- calls upon the kernel to read the **gzn** download code file
- copies the file into the controller's RAM
- when the download is complete, issues a forced call to start the downloaded code into execution
- performs a SYSGEN operation on each controller after the download.

In lines 13 – 65, the controllers driven by this device are initialized. This includes computing addresses used to pass data between the kernel and the device (lines 20 – 23), sending a RESET request to each controller (line 32), and waiting for an acknowledgement that the reset has been completed (lines 39 – 56). The driver uses the **delay**(D3X) function when waiting for the `RESET COMPLETE` message; it is important that the driver wait for this message with some mechanism that will not hang the system if the device is not responding.

In lines 66 – 102, the downloaded code is read into a buffer with the 3B4000/3B15 kernel function **drv_rfile**(D3X). The input is a pointer to an object file structure. This function will return a buffer address and a buffer size in the download file structure. The **open_close** element (line 98) indicates if the file should be opened and read (0) or closed (1). If a problem is encountered during the download process, an appropriate code is written to **u.u_error**; the "fall-back" mode (lines 69 – 87) is to continue on to the SYSGEN and let the controller come up with the resident firmware.

In lines 92 – 97, the driver resets the base address that it cleared for the pumpfile disk operation. The driver then moves the pumpcode from the kernel-allocated buffer to Controller memory (lines 98 – 99) and frees the buffer (lines 100 – 128). The device firmware may do this rather than the driver.

Prior to the start of operation, the driver communicates with the controller through a temporary stand alone command block (SACB) which is at a previously agreed upon address on the controller. To start the downloaded code running, the SACB is constructed then copied over the Local Bus a word at a time into the controller's memory. The controller is signaled to examine the SACB when the driver sets a bit in the board's Control and Status Register (CSR) to raise a Program Interrupt Request (PIR 1).

Now the driver waits for the SACBCMD flag to be reset by the interrupt handler (lines 132 – 150). If this does not happen within a "reasonable" period of time, an error message is written to the console and error log. This example ignores the failure and assumes that the device can be run from code resident on the board as a fall-back.

To initialize the contents of controller's sysgen data block, the driver puts information into the SACB for the sysgen request. This information would include such things as the addresses of the job request and completion queues and their sizes, along with any other information needed to establish communications between the driver and the controller. To do this, construct a temporary SACB, then copy it into the controller's memory over the Local Bus a word at a time. The word size is determined by the device, not the CC.

```
 1 gznstart()
 2 {

 3          struct cic_wcsr      *wcsrp;  /* write pointer to CSR */
 4          struct cic_rcsr      *rcsrp;  /* read pointer to CSR */
 5          struct pir32      *pirp;      /* write pointer to PIR */
 6
 7          int   delcnt;  /* intermediate delay cntr */
 8          int   ctlr;  /* controller counter */
 9          int   port;  /* port counter */
10          int   cnt;   /* transfer counter */
11          register char  *bufp;   /* Ptr to allocated buffer */
12          register char  *gznp;   /* Ptr to download memory */

13 /*   Controller Initialization */

14 /*   Initialize all controllers detected during boot */
15      for(ctlr=0; ctlr<gzn_cnt; ctlr++)
16      {
17 /*
18  *   Compute addresses of importance
19  */
20           wcsrp = (struct cic_wcsr *) (BIOADDR|OCSR);
21           rcsrp = (struct cic_rcsr *) (BIOADDR|OCSR);
22           sacbp = (unsigned short *) (BIOADDR|OSACB);
23           pirp = (struct pir32 *) (BIOADDR|OPIR);
24 /*
25  *   At this point set up any pointers needed for the Stand
26  *   Alone Control Block (SACB).

27  *   At this point the driver should contain code to initialize
28  *   data structures for the current controller and for each port
29  *   on this controller.
30  */

31 /*   Send RESET request to controller */
32           wcsrp->req_reset = SET;

33 /*   Allow CSR to be cleared by board from RESET request */
34           for(delay = 0; delay < DELAYMAX; delay++);
```

Figure 5–4 Initialization Routine 3B15/3B4000 Intelligent Device, part 1/5

```
35  /*   Wait for RESET COMPLETE to be set in controller's CSR
36  *    Look occasionally so as to not put unneeded traffic on
37  *    the bus
38  */
39            delay = 0;
40            TIMEDOUT = RESET;
41            while ((rcsrp->rcsr3 & RESET_COMPL) != SET)
42            {     if (delay < DELAYMAX)
43                  {     for (delcnt=-512; delcnt!=0; delcnt++)
44                        {     if((rcsrp->rcsr3 & RESET_COMPL) == SET)
45                                    break;
46                        }
47                        delay++;
48                  }
49                  else
50                  {
51                        cmn_err(CD_WARN,
    "GZNORPL %d: Reset timed out", ctlr );
52                        TIMEDOUT = SET;
53                        break;
54                  }
55            }

56            if(TIMEDOUT == SET)  /* check for reset timeout */
57            {
58  /*
59  *               At this point, take any action needed when a dead
60  *               controller is encountered.  Usually, all that can
61  *               be done is to mark it out of service, and avoid
62  *               using it during normal operations.
63  */
64                  continue;       /* Go on to next controller */
65            }
```

Figure 5–4 Initialization Routine 3B15/3B4000 Intelligent Device, part 2/5

```
66 /*   Clear the base io address to do the disk read. */
67           clearbaseio;

68           pmpfile.open_close = 0;
69           if (drv_rfile(&pmpfile))
70           {          /*   Kernel Failed to read pumpfile */
71                switch (u.u_error)
72                {
73                  case  ENOENT:
74 /*                  Do processing needed for missing pumpfile */
75                     break;
76                  case  EIO:
77 /*                  Do processing for read error on pumpfile */
78                     break;
79                  case  ENOMEM:
80 /*                  Do processing for insufficient main memory to
81                        read pumpfile */
82                     break;
83                  default:
84 /*                  Do processing for non-of-the-above error */
85                     break;
86                }
87                u.u_error = 0;  /* Reset error */

88                baseio(gzn_addr[i]);
89           }
90           else     /* Successful Read of GZN Pumpfile */
91           {
92                baseio(gzn_addr[ctlr]);
93
94                gznp = (char *)((long) BIOADDR | (long) GZNRAMADR);
95                bufp = pmpfile.buffer_addr;
96                for (cnt=0; cnt < pmpfile.buffer_size; cnt++)
97                     *etcp++ = *bufp++;
98                pmpfile.open_close = 1;
99                drv_rfile(&pmpfile);

100 /*   Set a flag that is cleared by gznint() to show completion */
101  */
102               SACBCMD = SET;
```

Figure 5–4 Initialization Routine 3B15/3B4000 Intelligent Device, part 3/5

```
103 /*
104  *   Set the PIR 1 bit in the controller's CSR to signal the
105  *   controller that a command is now available in the SACB
106  */
107                 pirp->pir01 = SET;

108                 delay=0;
109                 while (SACBCMD == SET)
110                 {
111                       if (delay < DELAYMAX)
112                       {
113                             for (delcnt = -512; delcnt != 0; delcnt++)
114                             {
115                                   if (SACBCMD != SET)
116                                         break;
117                             }
118                             delay++;
119                       }
120                       else
121                       {
122                             SACBCMD = FAIL;
123                             cmn_err(CD_WARN,
    "GZNORPL %d: Forced Call time out",
124                                         ctlr);
125                             break;
126                       }
127                 }
128           }

129           if (SACBCMD == FAIL)
130                 cmn_err(CD_NOTE,
    "GZNORPL %d: Controller in fall-back mode",
131                       ctlr);
```

Figure 5–4 Initialization Routine 3B15/3B4000 Intelligent Device, part 4/5

```
132           SACBCMD = SET;           /* set completion wait flag */
133           pirp->pir01 = SET;       /* set SACB command request pir */
134           delay = 0;               /* reset delay counter */
135           while(SACBCMD == SET)    /* wait for sysgen to complete */
136           {
137                if(delay < DELAYMAX)
138                {
139                     for(delcnt = -512; delcnt != 0; delcnt++)
140                     {
141                          if(SACBCMD != SET)    break;
142                     }
143                     delay++;
144                }
145                else
146                {
147                     SACBCMD = FAIL;
148                     break;
149                }
150           }

151 /*   Check for valid SYSGEN */
152           if(SACBCMD == FAIL)
153           {
154                cmn_err(CD_WARN, "GZNORPL %d: Failed SYSGEN", ctlr );
155                continue;     /* go on to next controller */
156           }
157      }
158 /*   Clean up a bit before returning */
159      clearbaseio;
160 }
```

Figure 5–4 Initialization Routine for 3B15/3B4000 Intelligent Device, part 5/5

Chapter 6: Input/Output Operations

Contents

Introduction

The main work of most drivers is moving data between user space and a device, usually with an intermediate transfer into kernel memory. This chapter provides the following information:

- General information on data transfer methods between the kernel and devices, and between user space and the kernel.
- Detailed information on block data transfer methods including information on character or physical I/O for a block device. This section assumes some familiarity with the header files and data structures discussed in chapter 4.
- Detailed information on character data transfer methods including information on buffered and unbuffered character I/O, and on allocating local driver memory. This section assumes some familiarity with the header files and data structures discussed in chapter 4.
- Detailed information on creating a private buffering scheme.
- Additional information on processor-specific memory management facilities.
- Additional information on scatter/gather I/O implementations.

Driver and Device Types

The UNIX kernel requires that all devices be classified as being *character*-access or *block*-access devices and that all drivers be of either a block or a character type. The terms block and character technically refer to the method used for data transfer. A block-access device transfers data one block at a time, using a cache of buffers the system maintains for data transfers. Special device files for block-access devices have a "b" in the first position of the file's permissions field.

Devices identified as character-access are basically devices that use any method other than the system buffer cache for transferring data. Some character-access devices transfer data one character at a time using `clists`(D4X), which are themselves a form of kernel buffering. The TTY line discipline (see Chapter 7) provides functions that do most of the `clist` manipulation for devices that require character processing such as terminals. STREAMS incorporates another character-access buffering scheme that should be used for most new communications drivers.[1] Other character-access device drivers may need to set up their own kernel buffering scheme, and transfer data in whatever unit that buffering scheme uses, or use local driver data space to buffer data being transferred between user address space and the device. Special device files for character-access devices have a "c" in the first position of the mode field.

Both block and character access devices can also use "raw", or unbuffered, data transfer schemes, although their implementations are different. Raw I/O is the movement of data directly between user address space and the device and is used primarily for administrative functions where the speed of a specific operation is more important than overall system performance. Character devices implement raw I/O through the **copyin**(D3X) and **copyout**(D3X) functions. Raw I/O is appropriate only for character devices such as line printers and some networking devices where the administrative software provides the capability to restart after an error.

Block access devices (such as a disk or tape) implement raw I/O using using the **physio**(D3X) function. The **physio** function locks the data in user address space (so it cannot be paged out) then transfers data directly between user address space and the device. Block-access devices supporting raw I/O must have both a block and a character special device file.

1. See Chapter 1 for ordering information for STREAMS documentation.

Data Transfer

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

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

The choice of which method to use depends on the type of the device, how much intelligence it supports, and the system utilities that will access it. Many transfers of data between user space and the device require an intermediate transfer of the data into the kernel memory.

Driver code should always use the function calls listed in Section D3X of the reference pages (especially **copyin** and **copyout**) for the actual data movement. These functions handle most of the memory management tasks that are required. The driver code must also validate the device number, handle errors that may occur during the transfer, and synchronize the software with the hardware event.

Whether a driver uses a private buffer or a system buffering scheme, every driver should be written with the finite nature of the machine in mind. Space used for buffering and local driver memory is taken away from memory that might otherwise be used for processes, so intense buffer use by a driver can reduce the performance of others drivers, or require that more memory be devoted to buffers. If more memory must be allocated to buffers, this decreases the memory available for user processes.

The discussion of data transfer in drivers has two facets: the driver's interaction with the operating system and the driver's interaction with the device.

Data Movement Between the Kernel and the Device

Data transfer methods between the kernel and the device are dependent upon the devices themselves. Some devices require the CPU to instigate all data transfer, while others can perform data transfers without the aid of the CPU. The details of a device's I/O scheme are always defined by the device, and so each device must be studied to determine precisely what kind of I/O scheme it supports.

In general, I/O devices can be separated into two main classes according to the way in which they transfer data to and from kernel memory

- programmed I/O devices that require the CPU to transfer data one byte or word at a time using a single input or output instruction to perform the data transfer

- direct memory access (DMA) devices that have the intelligence to perform the data transfer themselves and free the CPU to perform other tasks

For devices of the first class, the CPU transfers one byte or word of data by means of a specific instruction to or from a fixed register in memory to the device. Interrupts from the device control the timing of the data transfer. These types of devices are typically slow devices such as interactive terminals and older model line printers.

Devices that support DMA can transfer large amounts of data while freeing the CPU to perform other tasks. To initiate a DMA transfer, the CPU typically writes a base address and byte or word count defining the size of the block to be transferred to a previously allocated set of memory addresses. These addresses are referred to as the device's Control and Status Registers (CSR). The CPU then sets a bit in the device's CSR indicating that the transfer can begin. The device then performs the actual block transfer. When the data transfer is complete, the device sets a bit in its CSR indicating the transfer is complete, then issues an interrupt. Devices that support DMA are typically newer model character devices, and high speed block devices such as disks and tapes. Most devices supported by the computers discussed in this book utilize DMA I/O transfer schemes.

The characteristics of the DMA device itself determine how the driver is coded to do this transfer. The more complicated the device, the more memory addresses are allocated for the device's CSR. For example, a very simple device, such as a line printer, may have as few as two registers in memory: a status register and a buffer register. Characters are moved into the buffer register as long as a READY bit in the status register is on. When an interrupt is received from the device and the READY bit goes off, characters are held until the READY bit is turned on again. All the driver has to do is monitor and change the status register bits to effect the I/O transfer between memory and the device, and provide an interrupt routine.

A more sophisticated device, such as a disk controller, may have many registers each storing status information about specific subdevices including error logging. One register may contain a code for the type of I/O operation to be performed, while additional registers may contain the address location in memory where the data is to be moved to or from, the disk address, and a byte or word count. The intelligence on the board handles the details of the I/O transfer. The driver manages an internal queue of buffers using a private or system buffer scheme through its **read**, **write**, and **strategy** routines, and provides an interrupt routine for handling device interrupts.

These devices typically transfer large amounts of data, organized by page (2K bytes) or segment (128K bytes). If the device is equipped with DMA hardware, it may also provide a facility for handling I/O operations on a chained list of pages called a DMA list. Using this facility, the driver can transfer several pages of data at once rather than returning after each page transfer. The DMA list facility is discussed in the next section.

DMA Lists

Each write or read operation can transfer up to 2K bytes, or one page. So, to write 8K bytes of information, the driver actually executes 4 separate **write** requests. If the device has the requisite intelligence, you can do such a transfer more efficiently by setting up a DMA list, which allows the driver to transfer all 8K bytes to the device with one request. The DMA list organizes the

information into 4, 2K byte pieces, each of which has a pointer to where the data is in physical memory and a pointer to where the next piece is. After transferring one piece, it immediately begins the transfer of the next piece rather than return to the driver. Usually the board firmware is coded to handle this, in which case the actual registers, data, and control information all reside on the controller or device and the board firmware handles the virtual-to-physical translation. The kernel driver typically points the controller at the mapping structure and allows the controller to handle all translations required as well as the transfer itself.

The DMA transfer can be done without a DMA list. In this case, the driver keeps the data and control information in its own local area of memory. Data can be transferred between the device and kernel memory one byte at a time or DMA circuitry on the device can be used to copy larger pieces of data.

Data Movement Between the Kernel and User Space

Drivers moving data between kernel and user space can use either an array of private data storage in the driver's local area, a buffering scheme provided by the UNIX system, or a private buffering scheme. Private data storage can be used for character drivers that need to store small amounts of data. Memory is allocated through kernel memory allocation functions. These functions are described in ther "Allocating Local Memory" section of this chapter.

The following buffering schemes are provided by the UNIX system:

- the system buffering scheme defined in *buf.h* for block access operations
- the `clist` buffering scheme defined in *tty.h* for character access operations
- the STREAMS[2] buffering scheme for character access operations

The system buffering scheme uses a cache of preallocated kernel buffers called the *system buffer cache*. The system buffer cache is defined in the *buf.h* file. This file also declares a structure called `buf` which defines the fields contained in the buffer header (see Chapter 4). Block driver **strategy** routines receive a pointer to a buffer header through the *bp* argument. The buffer header defines all the information needed to perform the data transfer including the address where the data is to be transferred to or from and the amount of data to be transferred. The "Block Device Data Transfer Methods" section of this chapter discusses the use of the system buffering scheme in detail.

The `clist`(D4X) buffering scheme is provided by the TTY subsystem as a method of buffering character I/O. The `clist` buffering scheme is most frequently used with TTY line disciplines which provide functions for the management of `clists`. `clists` can also be used independently with a set of `clist` specific kernel functions. Chapter 7 of this book and the "Character Device Data

2. See Chapter 1 for ordering information for STREAMS documentation.

Transfer Methods" section of this chapter discusses the use of `clists` and the TTY subsystem in more detail.

Private buffering schemes can also be implemented, however they should only be created when necessary as they increase the size of the driver substantially. See the "Private Buffering Schemes" section of this chapter for more information.

Data Transfer Restrictions

The memory management scheme of the UNIX operating system does impose certain restrictions on drivers that transfer data between devices. Although the virtual memory block of storage for the data that is being transferred is contiguous in virtual memory space, it will be disjointed in the actual physical memory spectrum. The largest amount of physically-contiguous memory is one page. So, if the driver is going to pass 5K bytes of data to the controller for output, the driver will have to control where the page boundaries fall. To do this, make transfer sizes a multiple of 2K, aligned on 2K boundaries. Buffered I/O does this automatically, since buffers are preallocated and do not get faulted. Direct user/device transfers (raw) for block devices are managed by the **physio**(D3X) function, which handles the user data space schematics.

Block Device Data Transfer Methods

Drivers for block-access devices use two data transfer methods: block I/O and character or raw I/O. Block I/O uses the system buffer cache as an intermediate data storage area between user memory and the device. Character or raw I/O bypasses the system buffer cache and transfers data directly between user memory and the device using the **physio**(D3X) kernel function.[3]

Both block and character-access operations use the `buf` structure declared in the *buf.h* header file, but do so in different ways. For block-access operations, the buffer header is directly associated with a specific address in the system buffer cache. For character-access operations, buffer headers are taken from a separate pool of buffer headers called the physical I/O buffer header (PBUF) pool. These buffer headers are defined by the `buf` structure, but are associated with locked-in areas of user address space instead of addresses in the system buffer cache. The following diagram illustrates block I/O (Method 1) and character I/O (Method 2) on a block-access device:

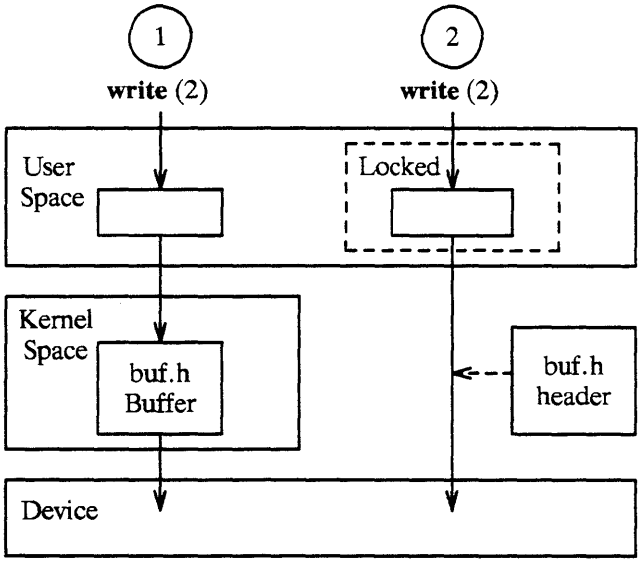


Figure 6–1 Two Methods of I/O Transfer (Block)

3. Character or raw I/O for block devices is also referred to as *physical I/O*.

Method 1 illustrates block-access to a block device. The system buffer cache is used to manage the actual read/write operations that move data between user address space and the kernel and between the kernel and the device. Your driver **strategy**(D2X) routine needs to define how to start and end the I/O operation, and frequently needs to maintain a private job request queue for each device. The kernel calls the **strategy** routine with the *bp* parameter which points to the *buf.h* buffer header containing all the information about the I/O operation.

Method 2 illustrates raw-access to a block device. The user address space for the data is locked in core, then the data transfer is done directly between the device and user address space using a buffer header extracted from the PBUF pool to control the operation. Your driver must include **read** and **write** routines which call the **physio**(D3X) function, and a **strategy** routine. The **physio** function calls the **strategy** routine as a subordinate routine to the **read** or **write** routine and passes it the *bp* parameter. The *bp* parameter points to the buffer header allocated for the data transfer.

The following sections discuss these two methods of block-access data transfer in greater detail.

The System Buffering Scheme

A block-access device uses block I/O, where data is read from or written to a device in units of a buffered block. On the 3B2 computer and SBC, a buffer is 1024 bytes; on the 3B15 and 3B4000 computers a buffer is 2048 bytes. Block I/O uses the *system buffer cache*, which has a tunable number of buffers and buffer headers (NBUF) and a tunable number of hash slots for the buffer cache (NHBUF). Each buffer has a buffer header associated with it that holds the control information about the buffer such as what block and what file system this data came from. This buffering scheme is defined in the *buf.h* header file.

When a block driver needs to move data between user space and the device, an appropriate number of buffers are made available to the device.

The data in a particular buffer remains in main memory until some other process needs a free buffer for some other I/O or until the driver clears the buffer with the `clrbuf`(D3X) function. Block I/O buffering has a number of advantages:

- **Data Cacheing** — The data remains in main memory as long as possible. This allows a user process to access the same data several times without performing physical I/O for each request. Since no physical I/O is done, the user process does not need to sleep while waiting for the I/O and thus runs more quickly.
- **Swapping Enabled** — If no buffering of data were done, a user process undergoing I/O would have to be locked in main memory until the device transferred data into or out of the user data space. Since there is a system buffer between the user data space and the device, the process can be swapped out until the transfer between the device and the buffer is completed, then swapped back in to transfer data between the buffer and user data space.

- **Consistency** — The operating system uses the same buffer cache as user processes when doing I/O with a file system, so there is only one view of what a file contains. This allows any process to access a file without worrying about timing.

Drivers that use the system buffering scheme must include the header file *sys/buf.h* and have a "b" under FLAG in the */etc/master.d* file. The `buf`(D4X) reference page lists the structure members that can be used and set by the driver.

The system buffering scheme allows drivers to transfer linked lists of data by using the **av_forw** and **av_back** members of the buffer header. Without this facility, an I/O operation would have to return after each buffer was transferred. For instance, when writing 6Kb of data, the driver would write 2Kb, return, write 2Kb more, return, and so forth. By using a linked list, the driver looks for the next buffer when it finishes transferring 2Kb of data, and only returns when the entire 6Kb are transferred. Note that the driver still performs three distinct operations, but it avoids the overhead of returning after each operation. With buffered I/O, no individual device/kernel transfer can exceed the size of a system buffer. It is not possible to allocate "contiguous buffers."

Utilizing this facility requires that the device itself have sufficient intelligence to handle its own linked list (defined in either pump code or operational code on the board). The firmware is coded to pick up the head of the linked list of buffers. The firmware driver translates the virtual address to a physical address goes to that physical location and writes the data, then goes to the physical location of the next buffer and so forth until the I/O transfer is complete. By moving this activity to the device itself, the kernel runs more efficiently.

The kernel handles memory management responsibilities such as controlling how segments and pages are broken down. The kernel-level driver must be aware of the scheme and make adjustments needed to accommodate the underlying device (such as presenting a job that crosses a segment boundary). The kernel-level driver must pass the virtual address, segment table address, and page table address to the firmware driver. The virtual-to-physical translation must be thoroughly tested by running extensive write/read operations and ensuring that what is read matches what was written. If the translation is wrong on a write operation, the driver writes invalid data; if the translation is wrong on a read operation, the driver may overwrite critical data in the kernel.

Using the System Buffering Scheme

For block drivers, kernel functions outside the driver itself control the actual data transfer operations. The driver itself utilizes five routines (See section D2X)

- **open** to open the subdevice
- **close** to close the subdevice
- **print** to report errors that happen during the actual data transfer operation
- **strategy** to validate job requests, manage the request queue, update controller and drive status, and generate work pending received interrupts
- **int** to report error status and release the buffers after the job completion interrupt is received

The **open**, **close**, and **print** routines are discussed elsewhere in this document. The following sections discuss the **strategy** and **int** routines.

Block Driver strategy Routine

The **strategy** routine is responsible for validating job requests, placing the request in the proper request queue (if the driver is using queues), updating the appropriate controller and drive status, and generating the work pending a programmed interrupt for the correct controller. All information to generate the job request is contained in the appropriate buffer header; the address of this buffer header is passed to the routine as an input argument.

The following validation checks are typically made:

- check for section boundary error
- check that subdevice is equipped (indicated in the **b_dev** member of the buffer header)
- check that the size (**b_blkno**) of the job request is reasonable

When validation tests in the **strategy** routine fail, the B_ERROR flag is set, an appropriate error code (usually ENXIO) is written to the **b_error** member, and **iodone**(D3X) is executed to terminate the operation. The kernel propagates **b_error** to **u_error** for the user-level process to see.

After the request is validated, an entry is made in the job request queue. This section of code should be protected from device-specific interrupts with an appropriate **spl***(D3X) function; the priority level is lowered after the request is sent to the controller for actual processing.

Then the buffer header is linked into the device work list. This is done using the **av_forw** and

av_back members of the buffer header.

If the driver is using job request queues, the job request, controller, and subdevice status data are updated next. When this is done, the job request is entered in the controller request queue. The buffer header address is used as the job id. The code checks whether **b_flags** is set to B_READ, and if so enters a read request; otherwise, a write request is issued. The **b_blkno** member of the header identifies the device-specific address to be read or written, and **b_bcount** specifies the number of bytes to be transferred, starting at the beginning of the buffer's **b_addr**.

At this point, the job is sent to the controller, and the priority level is returned to normal. For an example of a **strategy**(D2X) routine, see the driver in Appendix E.

Block Driver Interrupt Routine

When an I/O request is completed, or an error is detected, the device requests an interrupt. The CPU associates the device's interrupt with a driver **int**(D2X) routine. The driver's **int** routine identifies the type of interrupt and is passed a pointer to the buffer header in the system buffer hash list for that device.

If the interrupt is a normal job-completion interrupt, the driver's **int** routine relinks the **av_forw** and **av_back** members to set the next buffer transfer. Control of the data transfer is then given back to the device and the driver's **strategy** routine until the device requests another interrupt. When there are no more buffers to be transferred, the **int** routine issues a **wakeup**(D3X) for any processes that might be sleeping on the job request queue, then uses the **iodone** function to notify the user process that the I/O transfer is complete and to release the hash list of buffers.

If the device sends a failed-job interrupt, the **int** routine must set the **b_flags** member of the `buf` structure to B_ERROR; note, however, that it does not assign a value to the **b_error** member. Since such an error condition usually indicates some sort of hardware corruption, the error should also be written to the error log; **logberr**(D3X) is used for block-device errors.

Physical I/O for a Block Device

Most devices that use block-access also support raw or character I/O. Character I/O for a block device is referred to as physical I/O since data bypasses the system buffer cache and is transferred directly from the device to in-core user memory space. The advantage to physical I/O is that data can be transferred more quickly and in larger quantities than with the system buffer cache, and kernel overhead is reduced by eliminating buffer handling. However, because physical I/O actually locks down portions of user memory and prevents it from being paged, overall system performance is degraded. For this reason, physical I/O is used primarily for administrative functions where the speed of the specific operation is more important than overall system performance.[4]

A driver implements physical I/O for a block device through **read**(D2X) and **write**(D2X) routines. The character special device file for a block device indicates that the device supports physical I/O. The driver's **read** and **write** routines are then entered through the `cdevsw`(D4X) table. The **read** and **write** routines use the **physio** function to lock down the user memory and to call the driver's **strategy** routine. The **strategy** routine controls the actual I/O operation. Note that, in this case, the driver's **strategy** routine is called as a subordinate routine and not as a entry point routine.

The **physio** function allocates a free buffer header from a pool of physical I/O buffer headers set by the tunable parameter NPBUF. These buffer headers are defined by the `buf` structure, but do not point to a specific address in the system buffer cache. Instead, the data pointer is assigned the location in user memory where the data transfer should come from or go to. This location is determined from the **u.u_base** member of the user structure. The **strategy** routine then uses this buffer header to control the I/O operation.

The following is typical job sequence for a physical I/O **read** operation. A **write** operation is usually identical with the exception **b_flags** member of the `buf` structure is set to B_WRITE instead of B_READ. Figures 6-2 and 6-3 are example **read** and **write** routines for a disk driver using physical I/O. The line numbers included in the following job sequence refer to the Figure 6-2:

1. The user program issues a **read**(2) system call to the kernel of the form "read 10,240 bytes from *character-special-file* to *virtual-address-N*". The virtual address is a portion of user memory used to store user process data.

2. The kernel **read** routine started by the **read**(2) system call accesses the `cdevsw` table to call the driver's **read** routine. The `cdevsw` table is indexed by the internal major number; Chapter 3 describes how the operating system uses the MAJOR table to determine the internal minor number that corresponds to this device.

4. For example, when backing up a file system, one usually cares more about completing the backup quickly than maintaining optimal system performance during the time allotted for backup operations.

3 The driver's **read**(D2X) routine calls the **physck**(D3X) function to check that the range of blocks being read is legal, and returns a 1 if it is (lines 9-15).

4 The driver's **read** routine then calls the **physio** function to setup the I/O transfer (line 16). The **physio** function passes the address of the **strategy** routine, allocates a buffer header from the PBUF pool of buffer headers, and passes the buffer header the device number and the B_READ flag.

5 The **physio** function checks that all of the user pages in question are valid and have the appropriate read permissions, then locks the pages in user memory so they will not be paged out.

6 The **physio** function then calls the strategy routine and goes to sleep (using the **sleep**(D3X) or **iowait**(D3X) function) on the address of the buffer header until the I/O operation is completed. The functions used to synchronize hardware and software events are discussed in Chapter 9.

7 The **strategy** routine now controls the I/O. It checks the requests, queues it up, and does various conversions if necessary.

8 The **strategy** routine then starts the actual I/O operation. For example, it might put the read request into the control registers for the disk controller.

9 When the transfer is complete, the controller interrupts and the driver's **int**(D2X) routine is entered. The **int** routine uses the **iodone**(D3X) function to awaken the process that called the **physio** routine. The **physio** function then updates information on the `user` data structure, releases the buffer header, and eventually returns to the driver's **read** routine, which in turn returns to the kernel's **read** routine.

The following code examples are **read** and **write** routines from a sample disk driver:

```
1   dskread(dev)
2   register dev_t dev;
3   {
4         register unit; /* disk controller ID */
5         register unsigned char drv; /* disk drive ID */
6         register struct dskc *dskcp; /* disk controller pointer */
7         register struct dskpart *partpt; /* pointer to partition info */
8         register unsigned char part; /* drive partition */
9
10        unit = minor(dev);
11        dskcp = &dsk_dskc[unit>>5];
12        part = unit&07;
13        drv = (dev &030)>>3;
14        if ((partpt=dskcp->dsk_part[drv]) == NULL)
15              u.u_error = ENXIO;
16        else if (physck(partpt[part].nblocks, B_READ))
17              physio(dskstrategy, 0, dev, B_READ);
18  }
```

Figure 6–2 Disk read(D2X) Routine using Physical I/O

```
1   dskwrite(dev)
2   register dev_t dev;
3   {
4         register unit; /* disk controller ID */
5         register unsigned char drv; /* disk drive ID */
6         register struct     dskc *dskcp; /* disk controller pointer */
7         register struct dskpart *partpt; /* pointer to partition info */
8         register unsigned char part; /* drive partition */
9
10        unit = minor(dev);
11        dskcp = &dsk_dskc[unit>>5];
12        part = unit&07;
13        drv = (dev &030)>>3;
14        if ((partpt=dskcp->dsk_part[drv]) == NULL)
15              u.u_error = ENXIO;
16        else if (physck(partpt[part].nblocks, B_WRITE))
17              physio(dskstrategy, 0, dev, B_WRITE);
18  }
```

Figure 6–3 Disk write(D2X) Routine using Physical I/O

The **physio** function requires four arguments: *strat*, *bp*, *dev*, and *rwflag*. The **physio** function examples in the **read** and **write** routines provided above supply the standard values for those arguments:

- The *strat* argument is typically the address of the driver's **strategy** routine. In some cases, however, the routine called is a subroutine that performs a subordinate activity, such as calling the **dma_breakup**(D3X) function. The subroutine then calls the driver's **strategy** routine.
- The *bp* argument is the address of the buffer header. The safest way to invoke the *bp* parameter is with a null parameter; the **physio** function then assigns a buffer header internally. The **physio** function expects that any buffer header passed in corresponds to that defined in *sys/buf.h*.
- The *dev* parameter is the device number.
- The *rwflag* should be either B_READ or B_WRITE according to the operation.

Character Device Data Transfer Methods

Any device that supports only character-access is considered a character-access device. Unlike block I/O transfers that rely exclusively on the system buffer cache, there are many possible methods of implementing character I/O. It is important to know precisely what the device can and cannot do for you. The following factors must be considered:

- How much intelligence the device controller supports.

  Many character devices support DMA and can control their own I/O requests. Others can only perform one I/O operation at a time and require the CPU to control their I/O. Some character devices can even supply their own protocol requirements. Others need protocol packages supplied by the UNIX operating system, such as `tty` line disciplines.

- How much memory the device controller supports.

  Some character devices support DMA and are very intelligent, however, they may only support a small amount of local memory. Devices of this type may require additional kernel buffers.

- How much data is to be passed in a single I/O request, and how frequently requests are going to be made.

  Decisions as to the size of the buffers to be used depends upon the amount of data that is to be transferred.

In general, there are three possible schemes for doing I/O transfers for character-access devices: direct data transfer between the device and user space data buffering in memory allocated by the driver, data buffering in the kernel using a private buffering scheme, STREAMS[5] or the `clist`(D4X) buffering scheme. The following diagram illustrates these three character transfer schemes:

5. See Chapter 1 for a list of suggested STREAMS documentation.

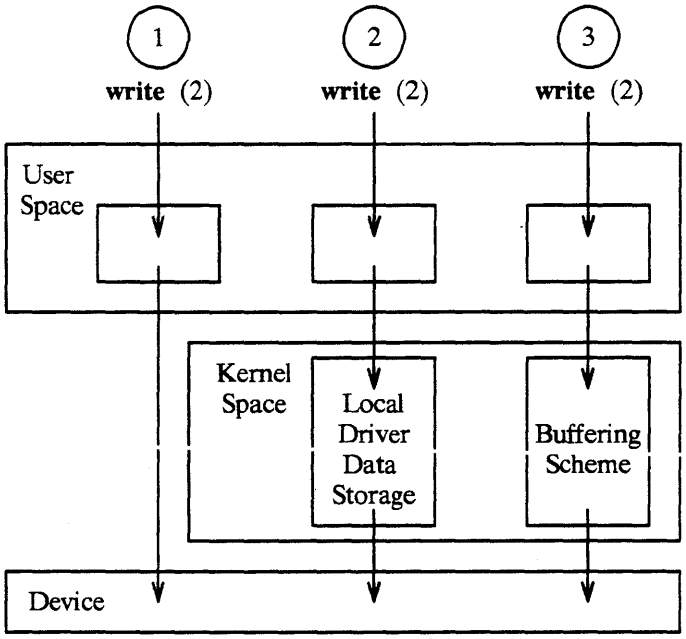


Figure 6–4 Three Methods of I/O Transfer (Character)

The operating system leaves most of the implementation decisions for character devices to the writer of the driver routines; you will need to select and implement the data transfer scheme that is most appropriate for your device. The following is a list of some general guidelines:

- Direct data transfer between the device and user space is most appropriate for devices that allow a restart after an error, such as network and printer devices.
- Use either STREAMS or `clists` for kernel buffering of asynchronous character I/O operations that happen frequently. Using system supplied buffering schemes reduces the kernel overhead.
- Private buffering schemes should be used only when absolutely necessary, since they use more memory and may be difficult to port to new machines and new UNIX System releases.

The following sections discuss buffered and unbuffered I/O schemes in more detail.

Buffered Character I/O

Most character device I/O is asynchronous, and so most character device drivers buffer data when passing it to and from the device. When reading, the driver must receive the data from the device in a read buffer, then copy the data from the buffer to the user process's local buffer. When writing, the driver must copy the data from the user process's local buffer into a write buffer, then transmit the data from the buffer to the device.

The TTY subsystem provides semantic processing of asynchronous character I/O, and a character buffering scheme called the `clist`(D4X) scheme. The `clist` buffering scheme is almost always used with TTY line disciplines, although `clists` can be used alone with `clist` specific kernel functions. The benefit of using the `clist` buffering scheme is that the pool of buffers, called `cblocks`(D4X), is allocated automatically when the system is initialized. However, the size of a `cblock` is 64 characters and cannot vary. Therefore, when moving small amounts of data, it may be more efficient to use memory that is allocated locally by the driver using memory allocation routines provided by the kernel. The next section discusses the use of these functions. The TTY line disciplines, the `clist` buffering scheme, and `clist` routines are discussed in detail in chapter 7.

Private buffering schemes that can range in complexity from a locally declared structure, to a module of separate memory initialization, allocation, and deallocation routines. The "Locally Allocated Memory" section discusses the allocation and management of small amounts of memory by the driver. The "Private Buffering Schemes" section discusses the types of routines and functions used to create a private buffering scheme.

Unbuffered Character I/O

Unbuffered character I/O is the transfer of character data directly between user space and the device, or using a small buffering area declared locally by the driver. Unbuffered I/O may be appropriate for a simple programmed I/O device that does not have much memory on the controller, or for a very intelligent device that maintains its own buffering scheme. Drivers for networking and printer devices may use this method, since the administrative software enables a restart if an error occurs during data transmission.

The kernel provides several routines to move unbuffered data. The most useful of these routines are **copyin**(D3X) and **copyout**(D3X). The `copyout` function copies data blocks from the buffers allocated by the driver to user space. It accepts as arguments the address of the driver buffer, the address of the user buffer, and the number of bytes to be copied. The **copyin** function copies data blocks from user space to the driver buffers and accepts the same arguments.

Because **copyin** and **copyout** handle page-faulting, they should always be used for unbuffered character I/O between the kernel and user space. A page fault occurs when a process attempts to access data that has been paged out. User processes can weather page faults by going to sleep until the data is paged back in, but some kernel operations may not be able to sleep while waiting for memory management to fault in a page. If a function that cannot handle a page fault attempts to access the user buffer when the user buffer is paged out, the system will probably crash.

Allocating Local Memory

Character devices frequently require a portion of memory to buffer small amounts of data, or to store an image of the data in memory to use to recover from an error condition. For instance, the **msg** module (see Figure 5-2) allocates memory to use when passing messages between processes. Some drivers, such as the 3B15/3B4000 system error log driver, use local driver memory to store records of device errors until the error daemon writes those records to a disk file. Other drivers need local memory only for a short time, such as when downloading data from a disk file to the device.

The easiest and least demanding method of storing small amounts of data is to declare a private structure or an array within the driver for the driver's private use. If more memory is needed, driver's can allocate private buffer space from a space management map. A set of memory allocation, deallocation, and management kernel functions can be used to allocate memory pages or variable size blocks of contiguous memory for the private use of the driver. The map management functions are defined in the *map.h* header file.

Tables 6-1 and 6-2 describe these kernel functions and the character driver routines in which they are used:

Table 6–1 Memory Map Management Routines

Task	Method	Routine(D2X)
Initialize a private memory map.	**mapinit**(D3X)	**init** or **start**
Allocate space from a memory map	**malloc**(D3X)	**read/write**
Release map entries	**mfree**(D3X)	**init** and **read/write**
Wait for a free buffer	**mapwant**(D3X)	**read/write**

Table 6–2 Memory Page Allocation and Deallocation

Task	Method	Routine(D2X)
Allocate memory pages	Use lines in master file if the amount of memory required is configuration dependent. Otherwise, use **kseg**(D3X) or **sptalloc**(D3X) in driver code.	**init**, **start**, or **open**
Release memory	**unkseg**(D3X) or **sptfree**(D3X)	**read**, **write**, or **ioctl** if memory usage is for a special case.

The map itself is declared as a structure using the driver prefix in the form *prefix***map**. Memory is initially allocated for the map either by a data array defined in the driver's master file, or by the **kseg** or **sptalloc** functions in the driver's **init** or **start** routine. The space management map is used to administer the buffer in bytes. Therefore, if **kseg** or **sptalloc** are used to allocate the initial memory, the number of bytes per page must be computed using the **ctob**(D3X) (clicks to bytes) function.

A driver initializes the map by calling **mapinit**, to establish the number of slots or entries to the map, and **mfree** to establish the decimal number of buffers free for use. Figure 6-5 illustrates the following procedures:

- the map structure declaration (line 3)
- the use of **kseg** to allocate memory for the map including a panic message if enough memory cannot be allocated (lines 10-14)
- the use of **ctob** to compute the number of bytes in the pages allocated by **kseg** (lines 17-18)
- the use of **mapinit** to configure the total number of slots in the map, and **mfree** to configure the total buffer area in bytes calculated by **ctob** (lines 15-21)

```
1   #define XX_MAPSIZE     12          /* In terms of slots */
2   #define XX_BUFSIZE      4          /* In terms of pages */

3   struct map  xx_map[XX_MAPSIZE];    /* Space management map for */
4                                      /*     a private buffer     */
5   ...
6   xx_start()
7   {
8   register caddr_t bp;
9   register int bytes;
10  if  ((bp = kseg(XX_BUFSIZE) == 0) /* Allocate private buffer; if */
11  { /* insufficient memory, display message & halt system */
12      cmn_err(CE_PANIC, " xx_start: kseg failed for %d page buffer allocation",
13                                              XX_BUFSIZE);
14  } /* endif */
15  mapinit(xx_map, XX_MAPSIZE); /* Initialize space management map */
16                               /* with number of slots in the map */
17  bytes = ctob(XX_BUFSIZE);    /* Compute the number of bytes in  */
18                               /* the pages allocated by kseg     */
19  mfree(xx_map, bytes, bp);    /* Initialize space management map */
20                               /* with total buffer area it is to */
21                               /* manage                          */
22    ...
```

Figure 6–5 Initializing a Memory Map

The **malloc**(D3X) function is then used by the driver's **read** or **write** routine to allocate buffers for specific data transfers. If the appropriate space cannot be allocated, the **mapwant**(D3X) macro is used to wait for a free buffer and the process is put to sleep until a buffer is available. When a buffer becomes available, the **mfree**(D3X) function is called to return the buffer to the map and to wake the sleeping process (no **wakeup**(D3X) call is required). The **copyin**(D3X) and **copyout**(D3X) functions are used to move the data between user space and local driver memory. The device then moves data between itself and local driver memory through DMA.

Figure 6-6 illustrates the following procedures:

- The size of the I/O request is calculated and stored in the **size** variable (lines 10-11).
- While buffers are available, buffers are allocated through the **malloc** function using the **size** value (line 13).
- If there are not enough buffers free for use, the **mapwant** macro is called, and the process is put to sleep (lines 14-19). When a buffer becomes available, the **mfree** function returns the buffer to the map and wakes the process.

- The **copyin** function is used to move data to the allocated buffer (line 21).
- If the address passed to the **copyin** function is invalid, the **mfree** function is called to release the previously allocated buffer, and the `u.u_error` field is passed a return error code.

```
1   #define XX_MAPPRIO (PZERO + 6)
2   #define XX_MAPSIZE    12
3   #define XX_BUFSIZE  2560
4   #define XX_MAXSIZE  (XX_BUFSIZE / 4)

5   struct map  xx_map[XX_MAPSIZE];  /* Private buffer space map */
6          char xx_buffer[XX_BUFSIZE]; /* driver xx_ buffer area */
7    ...
8   register caddr_t addr;
9   register int     size;
10      size = min(u.u_count, XX_MAXSIZE); /* Break large I/O request */
11                                         /* into small ones         */
12      oldlevel = spl4();
13      while((addr = (caddr_t)malloc(xx_map, size)) == NULL) /* Get buffer  */
14      {     /* if space is not available, then */
15           mapwant(xx_map)++;  /* request a wakeup when space is    */
16           sleep(xx_map, XX_MAPPRIO);  /* returned. Wait for space; mfree */
17                                       /* will check mapwant and supply   */
18                                       /* the wakeup call.                */
19      } /* endwhile */
20      splx(oldlevel);

21      if  (copyin(u.u_base, addr, size) == -1) /* Move data to buffer*/
22      {                                /* If invalid address is found, */
23          oldlevel = spl4();
24          mfree(xx_map, size, addr); /* return buffer to map */
25          splx(oldlevel);
26          u.u_error = EFAULT;       /* and return error code */
27          return;
28       } /* endif */
```

Figure 6–6 Allocating Memory From a Memory Map

Private Buffering Schemes

Character drivers may allocate independent buffer pools, although you should only do this when necessary since this increases the size of the driver, and thus the size of the kernel.

There are three main considerations involved in creating a private buffering scheme:

- What sort of memory management scheme should be used, such as memory mapping
- What sort of buffer header should be used; coupled or uncoupled

  Buffers and buffer headers can be either coupled or uncoupled. Buffers that are coupled with their buffer headers must be of a fixed size and in a specified location. Buffers that are not coupled with their buffer headers can be anywhere in memory, as long as the buffer header is pointing to its location.

- What sort of list management scheme should be used

  A buffering scheme can use any standard list management scheme. The most common schemes are various combinations of doubly-linked and singly-linked; circular versus noncircular; and with or without heads.[6]

The functionality required determines the specifics of a private buffering scheme. The following sections describe the requirements for any buffering scheme.

Creating a Private Buffering Scheme

The most practical way to implement a private buffering scheme is to write a separate module defining the buffering scheme. This simplifies maintenance tasks and enables you to use the buffering scheme for more than one device. This module should include subordinate routines for initializing, allocating and deallocating free buffers and in-use buffers, as well as tracking and error-handling routines. Any buffering scheme must include the following:

- a header file
  The header file defines the buffer and its headers. The buffer header should include links as well as members that track the status of the buffer, including any error conditions that have occurred. It may be appropriate to use the `buf` structure defined in *buf.h*.

6. For more information on list management schemes, consult a general computer text such as Knuth, D.E., *The Art of Computer Programming*, vol. 1.

- a pool of free buffers.
  These may be defined in the */etc/master.d* file and allocated statically when the driver is initialized, or defined in the driver code and allocated dynamically, usually in the driver initialization routine.

- a set of lists used to manage buffers in different states (free, active, queues, and so forth)

- routines for moving buffers between lists (for instance, allocation of a free buffer, releasing a buffer, queueing a buffer for work, and so forth)

It is possible to dynamically allocate buffers based upon need, but this is usually very expensive if it occurs frequently. The overhead is significant, but it does reduce the amount of allocated memory. When a buffer is required only for device initialization or some other infrequent event, dynamically-allocated buffers may be useful. For buffers used for frequent events, statically-allocated buffers are usually the preferred implementation.

Header File

The header file for the buffering scheme should define the structures being used. This usually includes a structure that holds free buffers, a structure that holds buffers that are in use, and a structure defining a header that holds status and flag information pertaining to a given buffer. If the buffering scheme is a doubly-linked circular list, you may want to use the `buf`(D4X) structure declared in the *buf.h* file. In any event, the `buf` structure provides a good example of the members that should be included in a buffer header. The header file should also include a definition of any flags, status indicators, or special error codes used by the buffering scheme.

In addition to the data structures defined for this module, the *map.h* system header file must be included if the buffering scheme is managed by a memory map.

Master File

The master file for the module that defines the private buffering scheme should use the "o" and "x" in the FLAGS field and define the module's prefix in the PREFIX field; all other columns except the DEPENDENCIES/VARIABLES column are left blank.

The DEPENDENCIES/VARIABLES column should include tunable parameters that control the size of the buffer pool being allocated. For example, the sections that follow introduce a hypothetical buffering module named **qq_** used by a driver named "DDD". NDDDPORT is a tunable in the DDD driver that defines the maximum number of ports that can be controlled by a single DDD device. The **qq_** module uses this number to determine the number of buffers to allocate.

The **qq_** master file should include a comment that explains the algorithm used to determine the size of the buffer pool.

Private Buffering Scheme Routines

The code for allocating and deallocating memory, assigning and freeing buffers, and transferring data between user space and the kernel should be defined in separate subroutines, each of which should use a common prefix. Figure 6-7 summarizes the subroutines that have been created for the QQ buffering module. The same types of subroutines should be creating for any private buffering scheme.

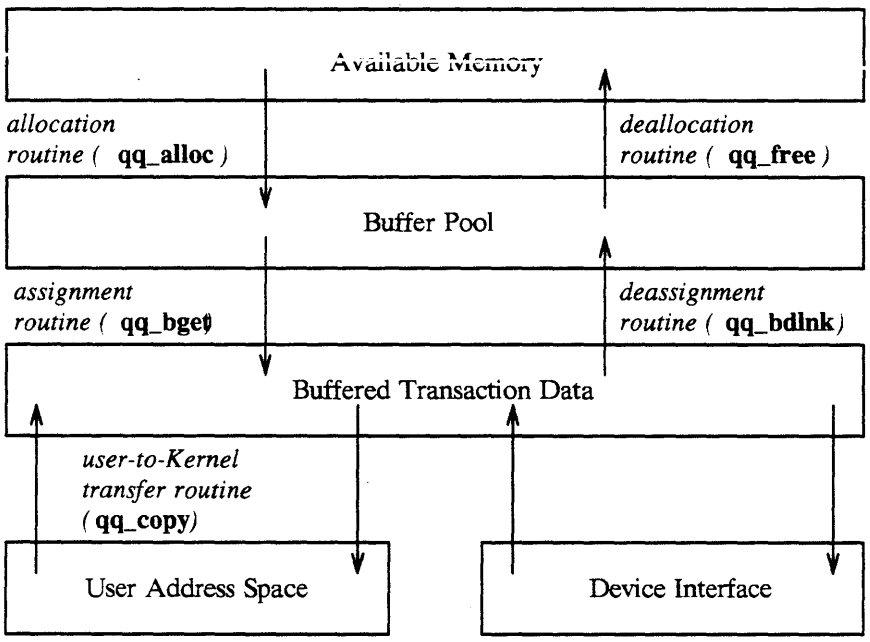


Figure 6–7 Routines Used for a Private Buffering Scheme

Memory Allocation Routine

The memory allocation routine (**qq_alloc**) creates a map for the pool of frree buffers that are available to drivers using the buffering scheme. The amount of memory allocated should be set as a variable that is indirectly modified by tunable parameters in the module's master file.

As in the locally allocated memory examples previously outlined, the **mapinit**(D3X) macro is used to initialize a memory management map in the format of *sys/map.h*, and **mfree**(D3X) to "free" the memory into the map (lines 19-20). The size of the buffer and the buffer's address are saved in `cnt` and `segp` (lines 21-22), and the free buffer descriptor pointer is initialized to NULL (line 23).

Note that the second argument to the **malloc**(D3X) function, *size*, is expressed using **ROUND(x)** operand that ensures that memory is allocated on a word boundary. In other words, if you ask to allocate three bytes, the system will actually allocate four bytes.

```
1  #define ROUND(X)    ((X+3) & ~3)
2    ...
3    mminit()
4    {
5         mapinit(mmmap, nmmd);
6         mfree(mmmap, mmdsz, mmd);
7    }
8    ...
9    int first_call;

11   qq_alloc(qq_bufp, nbytes)
12   register struct qq_buf     *qq_bufp;    /* Ptr to qq_buf structure */
13   int              nbytes;          /* Size to be allocated */
14   {
15          register char             *segp;
16          register unsigned         cnt;

18                  return(NULL);
19           mapinit(qq_bufp->qq_map, QQMAP);
20           mfree(qq_bufp->qq_map, cnt, segp);
21           qq_bufp->qq_bsz = cnt;
22           qq_bufp->segp = segp;
23           qq_bufp->freebdp = NULL;
24           return(nbytes);
25   }
```

Figure 6–8 Memory Allocation Routine

Allocating and freeing pages should be done very carefully; if it is done incorrectly, it can crash the system or corrupt user processes and the disk. Performance degradation may not show up until heavy loads are applied, and it may be intermittent.

Memory Deallocation Routine

The memory deallocation routine (**qq_free**) releases the memory mapped by a buffer header by first allocating all the memory in the map with **malloc**(D3X) (line 6), then releasing the block with **mfree**(D3X)[7] (line 12). A pointer is used with the **mfree** function to indicate which block of memory should be deallocated. The routine must first check whether the block is still owned (in other words, whether memory is still allocated out of the buffer memory map). If so, it should send a message to the console, then free the block in smaller pieces (lines 7-10).

```
1   qq_free(qq_bufp)
2    register struct qq_buf    *qq_bufp;    /* Ptr to qq_buf structure */
3    {
4           register int  i = 0;
5
6           if(malloc(qq_bufp->qq_map, qq_bufp->qq_bsz) == 0) {
7                   cmn_err(CD_WARN,"qq_free: Can't free block\n");
8                   for(i = QQBSZ; i; i >>= 1)
9                           while(malloc(qq_bufp->qq_map, i));
10                  i = -1;
11  }
12          mfree(ksegmap,qq_bufp->qq_bsz, qq_bufp->segp);
13          qq_bufp->qq_bsz = 0;
14          return(i);
}
```

Figure 6–9 Freeing Private Memory Blocks

7. **unkseg**(D3X) could be used rather than **mfree**.

Buffer Assignment Routine

The assignment routine (**qq_bget**) assigns an appropriate number of memory pages from the buffer pool to support the particular I/O transaction. The routine first checks that buffers are available; if not, it can either wait on the buffer header until a buffer is available (as in the example, lines 11-14) or return a 0 (zero) to indicate that all `map` entries are allocated. When a buffer is attached, the freelist header must be updated to reflect that this buffer has been removed (line 20), then return to the calling process that the buffer has been allocated (line 26).

```
1   struct qq_bd *
2   qq_bget(qq_bufp, nbytes, slpflg)
3   register struct qq_buf  *qq_bufp;      /* Ptr to qq_buf structure */
4   int             nbytes,                /* Size of buffer to get */
5                   slpflg;                /* Sleep flag */
6   {
7         register char            *addr;
8         register int             sps;
9         register struct qq_bd    *bdp;

10        sps = spl5();
11        while((bdp = qq_bufp->freebdp) == NULL ||
12              (addr = (char *)malloc(qq_bufp->qq_map, ROUND(nbytes))) == 0) {
13              if(slpflg)
14                      sleep((caddr_t)&qq_bufp->freebdp, QQSLP);
15              else {
16                      splx(sps);
17                      return(NULL);
18              }
19        }
20        qq_bufp->freebdp = bdp->d_next;
21        splx(sps);
22        bdp->d_size = ROUND(nbytes);
23        bdp->d_ct = nbytes;
24        bdp->d_address = nbytes ? addr : 0;
25        bdp->d_next = NOLIST;
26        return(bdp);
27  }
```

Figure 6–10 Moving a Buffer from the Pool

Buffer Deassignment Routine

The deassignment routine (**qq_brtn**) returns a buffer to the freelist after the operation is completed. The routine first checks that the address is not zero (line 7), frees the buffer with **mfree** (line 8), then links the buffer to the freelist. A **wakeup**(D3X) call is issued in case any processes are sleeping on the resource (line 12).

```
1   qq_brtn(qq_bufp, bdp)
2   register struct qq_buf   *qq_bufp;          /* Ptr to qq_buf structure */
3   register struct qq_bd    *bdp;           /* Ptr to bd to return */
4   {
5           register int     sps;

6           sps = spl5();
7           if(bdp->d_address && bdp->d_size)
8                 mfree(qq_bufp->qq_map, ROUND(bdp->d_size), bdp->d_address);
9           bdp->d_next = qq_bufp->freebdp;
10          qq_bufp->freebdp = bdp;
11          splx(sps);
12          wakeup((caddr_t)&qq_bufp->freebdp);
13    }
```

Figure 6–11 Returning a Buffer to the Pool

User-to-Kernel Transfer Routine

The private buffering scheme should include its own routine to move data between itself and the user address space. This routine can call the **iomove**(D3X) or **copyin**(D3X)/**copyout**(D3X) functions which handle page faults and update the `user` structure.

```
1   qq_copy(bdp, offset, cnt, rdwr)
2   struct qq_bd    *bdp;              /* Buffer desc. pointer */
3   int             offset,            /* Offset into data buffer */
4                   cnt,               /* Number of bytes to transfer */
5                   rdwr;              /* Read or write */
6   {

8           if(cnt == 0)
9                   return(0);
10          iomove((caddr_t)(bdp->d_address + offset), cnt, rdwr);
11          if(u.u_error)
12                  return(-1);
13          return(0);
}
```

Figure 6–12 Moving Data Between the Buffer and User Address Space

Kernel-to-Device Transfer Routine

The private buffering scheme may include its own routine to transfer data between the kernel buffers and the device. If the device supports DMA, it can be given the location (address) of the buffer along with some form of job request data structure. The device then handles the actual I/O operation. Less intelligent devices may require the CPU to perform the actual I/O transfer, in which case a specific routine must be written to facilitate the transfer.

Coding the Driver to use the Private Buffering Scheme

To write a driver that utilizes the private buffering scheme, the system entry point routines use a combination of the functions in Section D3X and functions that are routines in the module that defines the buffering scheme. The following list outlines the types of considerations:

- Header Files

  The driver code that accesses the private buffering scheme must include the header file for the buffering scheme as well as the *sys/map.h*, *sys/user.h* and *sys/errno.h* header files. If the buffering scheme is using an existing header file (such as `buf`(D4X)), include the appropriate header file (in this case, *sys/buf.h*).

- Driver Initialization Routine

  The driver's initialization routine (**init** or **start**) allocates the buffers for the private buffering scheme. It does this by calling the allocation routine (**qq_alloc**) then the deassignment routine (**qq_bdlnk**) to ensure that the buffers are actually free. The code should be written to handle the case where memory is exhausted by using **cmn_err**(D3X) to print a warning notice to the console and setting the **u.u_error** member of the `user`(D4X) structure to ENOMEM.

  Some drivers may choose to allocate a "starting pool" of buffers and use this until demand exceeds the size of the starting pool ("high-water mark"). It could then allocate more memory to enlarge the pool. After the pool is back to a certain free level ("low-water mark"), the extra memory would be released.

- Driver **read**(D2X) Routine

  The driver's **read** routine uses the assignment routines (**qq_bget** and **qq_emptq**) to assign buffers to this operation, the device-interface routine (either from the module code or the firmware driver) to move data from the device to the kernel buffer, and the user-to-buffer transfer routine (**qq_copy**) to move the data to the user address space. It then calls the deassignment routine, **qq_brtn**, to return the buffers to the buffer pool.

- Driver **write**(D2X) Routine

  The driver's **write** routine uses the assignment routine, **qq_bget**, to assign buffers to this operation, then calls the user-to-buffer transfer routine (**qq_copy**) to move the data from user address space to the reserved buffer. The **write** routine calls a subordinate routine to transfer the data from the buffer to the device. This subordinate routine should call the buffering scheme's kernel-to-device routine. When all the data has been transferred to the device, the driver's **write** routine calls the deassignment routine (**qq_brtn**) to return the buffer to the buffer pool.

Machine-Specific Memory Management Information

While the memory management schemes for the computers supported by this document are similar to each other, some machine-specific memory management facilities have been introduced to fully utilize the architectures of the various machines. These are discussed below.

The WE® 32101 Memory Management Unit

All computers supported by this book are based on the WE 32101 chip. *Maxicomputing in Microspace*[8] gives a full description of the WE 32100 chip, including the Memory Management Unit (MMU). This section provides some of the basic facts that are of particular interest to driver writers.

Each WE 32101 MMU has a cache for 32 segment descriptors and 64 page descriptors from previous translations.[9] Cached entries reduce translation time on subsequent references to the same segments and/or pages, since it is not necessary to access memory to read the translation table(s). Sections provide a convenient way to divide virtual address space into separately managed chunks. This is particularly valuable in maintaining a process's descriptor tables so as to lessen the chance that a table will grow so much that it must be moved. For example, since both user data and stack areas are expandable, if they were mapped within one section it might often be necessary to move all stack segment descriptors to make room for more data segment descriptors. Moving the user stack to a separate section minimizes this problem.

3B15 Dual MMU

The 3B15 computer and the 3B4000 Master Processor have dual MMUs. In essence, virtual memory is divided into eight separate sections, with each MMU handling four sections. This doubles the MMU on-board descriptor cache and the available sections.

The dual MMU hardware is implemented as follows:

- The two WE 32101 MMUs are accessed in memory-mapped peripheral mode at two discrete addresses: MMU0 is accessed at 22000 and MMU1 is accessed at 23000.
- Bit 29 of a virtual address is used by the hardware to select an MMU to perform the address translation. In this sense, bit 29 becomes the field used to select an SRAMA/SRAMB register set or section.

8. See Chapter 1 for ordering information.
9. A page is 2K; a segment is 64 pages, or 128K.

- Bit 29 is also still used by an individual MMU as the high-order bit in the Segment Select.

The use of bit 29 to select an MMU and the separate memory-mapped locations for the MMUs result in the following section/memory location mappings:

VA bits	31	30	29	=	Section	MMU	SRAMA/B address
	0	0	0	=	0	0	22600/22700
	0	0	1	=	1	1	23600/23700
	0	1	0	=	2	0	22604/22704
	0	1	1	=	3	1	23604/23704
	1	0	0	=	4	0	22608/22708
	1	0	1	=	5	1	23608/23708
	1	1	0	=	6	0	2260c/2270c
	1	1	1	=	7	1	2360c/2370c

Much of the work for utilizing the dual MMU is handled for drivers by the operating system. The *sys/immu.h* and *vuifile* recognize the dual MMU and the `user` structure has additional storage areas that hold SRAMA/SRAMB values. In addition, memory fault handling utilities on the 3B15 computer and 3B4000 MP handle faults generated by either MMU.

Because of the dual MMU, drivers that are doing virtual-to-physical translation must specify which part of memory is involved. For this purpose, 3B15 has the **getsrama**(D3X) and **getsramb**(D3X) functions that return the contents of the SRAMA and SRAMB registers based on the section id and address given. These macros should be used when the contents of an SRAM are to be used to perform any type of address conversion, since the kernel and hardware view of the location of memory management tables are totally different. The physical address of an MMU1 descriptor table as lowered by 0x8000 to meet hardware needs does not represent the actual table location known by the kernel and may, in fact, be an address less than that of the first physical page mapped by a `pfdat` structure.

The following example illustrates how driver code determines which MMU is being used.

```
long sid;       /* Temp storage for section id from virt add */
paddr_t psdtpt;       /* Pointer to top of segment descriptor tbl */
long psdtln;       /* length of sdt */
...
sid = (*(VAR *)&maddr).v_sid; /* get section id from virtual address*/
psdtpt = getsrama(sid);      /* get phys top of sdt */
psdtln = getsramb(sid);      /* get length of sdt */
```

Figure 6–13 Example of Accessing Dual MMU

Accessing Non-Local Memory on the SBC

On the SBC, the local memory pages on the CPU board are supplemented with non-local (VME) memory. Local memory has a physical address below 0x200000; VME memory has a physical address above 0x200000. To allocate non-local memory, ask for memory with the **sptalloc**(D3X) function. Check to see if it is local or VME by translating its virtual address to a physical address (use the **vtop**(D3X) function) and checking to see if it is local or VME. Using **kseg**(D3X) and **unkseg**(D3X) may also work.

The VME A24 address space on the SBC is limited to 16 MB. By using VME A32 space, you can get more memory if your device produces A32 address modifiers and you have a memory board that accepts A32. However, if your driver uses this, no other device in the system (except the CPU board) can produce an A32 address modifier and access that memory. This means that A32 memory cannot be used for normal activities such as process pages. In most cases, do not use the A32 memory for a driver.

Accessing Local Processor Memory on 3B4000 Adjuncts

On the 3B4000 computer, user-level processes are usually assigned to whichever processor has the least number of processes,[10] which maximizes the performance advantages of the multiprocessor architecture. Drivers, however, are located in the kernel of the processing element on which the hardware is located. Because the ABUS bootstrap process (see Chapter 4) configures each adjunct processing element individually, using a master file and an executable object file that are marked for the appropriate processing element, all that is necessary is to put these files under the appropriate */adj/pe#* directory (*/adj/pe#/etc/master.d and /adj/pe#/boot*), and, for software drivers, add an INCLUDE line to the */adj/pe#/etc/system* file and the driver will be part of the adjunct kernel.

10. This automatic assignment can be overridden with the **pe**(1) command or the **sysmulti**(2) system call.

Scatter/Gather I/O Implementations

A number of modern I/O boards (primarily disk controllers) support I/O schemes other than the traditional "move this one piece of data to this one location." These schemes are referred to as scatter/gather I/O implementations. Note that the term "scatter/gather" is used differently by different vendors, so that a board that is advertised as supporting such I/O operations may support any or all of the implementations discussed below. The following pages describe how to write a driver that utilizes these board capabilities.

Request Chaining

Request chaining is the capability of a device (such as a disk controller) to accept an array or linked list of individual I/O jobs from the CPU. The disk controller will execute all the jobs and give one completion interrupt at the end of the sequence.

A job is an operation such as "read block N to physical address X" or "read 5 blocks, starting at block N, to memory starting at physical address X".

Request chaining can only be implemented for boards that support such an operation. The driver code should then contain a private routine (based on **dma_breakup**(D3X) but given a different name) that passes an entire chain of requests to the **strategy**(D2X) routine rather than passing one page at a time. The driver functions can then operate on the whole chain of requests simultaneously, do all the checking and address translations, and give the whole chain to the disk controller.

Be sure that you have preserved the standard interface to the **strategy** routine. You may have to move the bulk of the **strategy** routine to a driver-specific routine and have both your version of **dma_breakup** and what remains of the driver's **strategy** routine call this driver-specific routine.

The controller may set a "done" bit in each request block as the request is completed, so that the CPU can peek at the list even before the job completion interrupt occurs. This is an optimization.

Multiple Copying

Multiple copying refers to the capability of a device to accept an I/O job that requires a one-to-many copy. Several identical copies of the data are written to multiple places. For instance, "write 1 block of data from address X to disk blocks M, N, and O" or "read block N from disk and copy it to addresses X, Y, and Z".

Note that multiple copying is different from multi-block transfer. Multi-block transfer is the ability to copy two blocks to one address X in one I/O request. Multiple copying is the ability to copy the same two blocks to different addresses, such as 0x100000, 0x700000, and 0x123450. This could be used, for example, to set up mirroring capabilities where the actual write operation is done to a mirror pseudo-device which then writes the same information to two physical devices.

Multiple copying can only be implemented for boards that have this capability.

Virtual DMA

Virtual direct memory access (DMA) is the ability to accept I/O jobs that contain virtual addresses rather than physical addresses. Each "job" would be of the form "read block N to virtual address X" or "read 5 blocks, starting at block N, to memory starting a virtual address X".

To support this implementation, the board must be able to translate virtual addresses into physical addresses, which means that the board's firmware must contain a basic subset of the memory management scheme, including the format of the memory management tables used by the MMU.

To utilize virtual DMA, create a private driver routine based on the **dma_breakup**(D3X) function. Since a virtual DMA board understands page boundaries and address translation, rather than breaking up the request the modified **dma_breakup** function can simply pass the entire request to the **strategy**(D2X) routine. Create another private routine that is based on the **iostart**(D3X) function but without the virtual-to-physical translation. Give the entire request to the board. You should not have to split up the **strategy** routine for virtual DMA I/O.

Some boards (such as the MCT 6020 on the SBC) have to be given a special copy of the MMU tables. You have two options for accomplishing this

- Create these special tables from the real MMU tables every time an I/O request occurs. This may hurt the performance of your driver but localizes the changes to your driver and enhances its portability.
- Create the tables once when the process is created and then keep them consistent with the MMU tables over the life of the process. This means that you must modify the kernel memory management functions for the device every time a page is paged out or created.

Chapter 7: Drivers in the TTY Subsystem

Contents

Introduction

This chapter describes the components of the TTY subsystem. The TTY subsystem is a collection of functions and the driver **proc**(D2X) routine that are used to transfer information character-by-character between a CPU and a peripheral device such as a terminal or printer. These functions are found in the *ttl.c*, *tty.c*, and *clist.c* source code files. These functions are also known as Common I/O (or CIO). Another frequently used term is *line discipline*. A line discipline is a set of functions that interprets the data received from a terminal to extract special characters such as the BREAK and the DELETE keys and moves data between a terminal and a user program. The TTY subsystem involves access of the `tty`(D4X) structure defined in *tty.h* and is described in this chapter.

A line discipline ensures a user program that

- Data received from a terminal is in the range of printable ASCII values, or if special processing is disabled, that the data is conveyed to the program exactly as entered (except for BACKSPACE , BREAK , DELETE , and "Quit").
- Characters sent from the user program are correctly displayed on the terminal screen.

All of these concepts are explained in greater detail in the sections that follow in this chapter.

A wide range of devices exist for moving data character-by-character between a device and the host computer. Examples of these devices are

- terminals
- printers
- network handlers
- robots
- laboratory applications

Occasionally, these devices require drivers that convey the data from the device to a user program. These drivers typically interpret the characters that are received from the device before they are delivered to the user program. This is especially true in devices using some sort of keyboard that allows data flow to be interrupted or terminated. For these applications, the driver must rely on routines to initiate special processing requirements when interrupt or flow control keys are pressed.

The UNIX operating system TTY routines provide character interpretation (called canonical processing). The characters which are processed include, the erase character, the kill character, the end-of-file character, characters preceded by a backslash, and upper/lower case presentation characters.

Canonical processing means translating the actual characters typed to produce what the user intended. For instance, if the ERASE character is represented by # and the raw input is

H u # e l l o

the canonical output is

```
Hello
```

Data is received from the terminal keyboard and placed in the **t_rbuf** receive buffer. **ttin**(D3X) does initial character processing and moves valid data to the **t_rawq** raw character queue. **canon** processes more characters and moves the valid characters to the **t_canq** canonical (processed character) queue. If characters are requested to be echoed to the screen, valid characters are placed in the **t_outq** output queue. Input characters, whether echoed or not, are then conveyed to the user program by **ttread**. **ttwrite** conveys characters from the user program to the **t_outq** output queue. **ttout** conveys characters from the **t_outq** output queue that are echoed or being sent from the user program to the **t_tbuf** transmit buffer. A terminal dependent output routine conveys the data from **t_tbuf** to the terminal's display. Figure 7-1 illustrates how characters are transferred between a terminal and a user program.

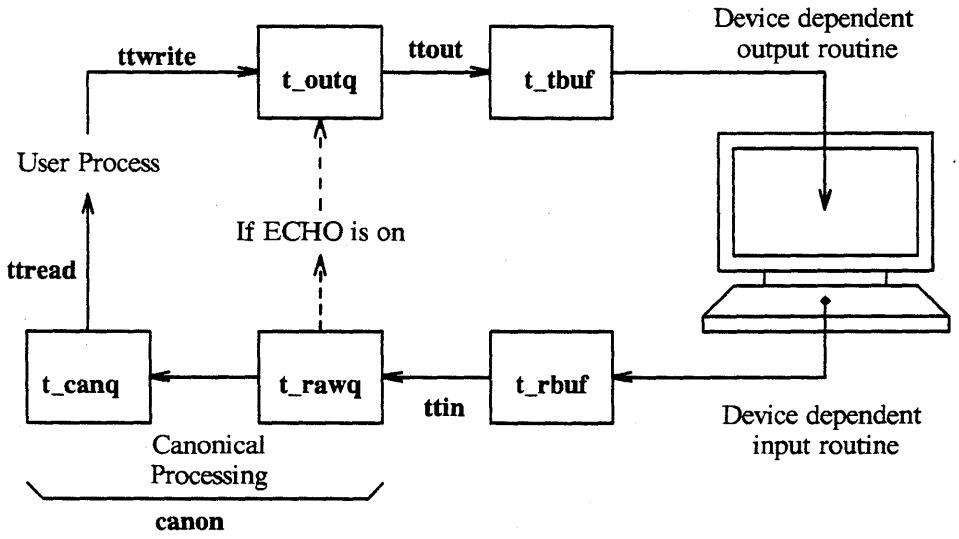


Figure 7–1 TTY Functions

In addition to the functions specified in the line switch table for interpreting characters, other support functions are provided as well. Figure 7-2 lists the Common I/O functions.

Function (D3X)	**Description**
canon(*tp*)	Evaluate characters and move data from **t_rawq** to **t_canq**
getc(*clp*)	Get a character
getcb(*clp*)	Get first character block
getcf()	Get free character block
putc(*c, clp*)	Put data on a character list
putcb(*cbp, clp*)	Link a character block to a character list
putcf(*cbp*)	Release a character block
ttclose(*tp*)	Close a character device
ttin(*tp, code*)	Get data from the device-dependent input routine
ttinit(*tp*)	Set a `tty` structure to default values
ttiocom(*tp, cmd, arg, mode*)	Process internal requests
ttioctl(*tp, cmd, arg, mode*)	Process internal requests
ttopen(*tp*)	Open a character device
ttout(*tp*)	Transfer data to the device-dependent display routine
ttread(*tp*)	Move input data to user process
ttrstrt(*tp*)	Restart data flow
tttimeo(*tp*)	Time function for **termio**(7) "TIME"
ttwrite(*tp*)	Take data from user process
ttxput(*tp, ucp, ncode*)	Put data into output queue
ttyflush(*tp, cmd*)	Release unneeded buffers
ttywait(*tp*)	Delay processing

Figure 7–2 Common I/O (CIO) Functions

Detailed information on the functions in Figure 7-2 is presented in Section D3X of the *BCI Driver Reference Manual*, referenced in Chapter 1.

Line Disciplines

A line discipline contains functions for opening, closing, reading, writing, input/output control, data receive interrupts, data transmit interrupts, and modem interrupts. Each of these activities is defined by individual members of the `linesw` (line switch) structure found in *conf.h*. The primary functions involved in writing a line discipline are: **canon**(D3X), **ttin**(D3X), **ttout**(D3X) and **ttxput**(D3X).

Currently, three line disciplines are defined; however, up to 256 are permissible. The **t_line** member of the `tty`(D4X) structure is the index into the line discipline switch table. A driver can access as many line disciplines as required. The line disciplines allocate memory for data buffering purposes for operations associated with the device (such as moving `cblocks`(D4X) from the free list to this `tty` structure) and implementing flow control. Flow control is the ability of the operating system to control the rate of data transfer between a device and the system. One example of flow control is CTRL-q and CTRL-s for starting and stopping screen displays.

Line disciplines are defined by placing information about a line discipline in the *kernel* master file. Figure 7-3 shows an example *kernel* master file.

```
 1   * Line Discipline Switch Table
 2   *   order: open close read write ioctl rxint txint modemint
 3   linesw (%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1%1)
 4   ={
 5   *     TTY ----------------
 6               &ttopen, &ttclose, &ttread, &ttwrite,
 7               &ttioctl, &ttin, &ttout, &nulldev,
 8   *     XT -----------------
 9               &nulldev, &nulldev, &nulldev, &nulldev,
10               &nulldev, &xtin, &xtout, &nulldev,
11   *     SXT ----------------
12               &nulldev, &nulldev, &nulldev, &nulldev,
13               &nulldev, &sxtin, &sxtout, &nulldev,
14   }
```

Figure 7–3 Example kernel Master File

The XT and SXT line disciplines consist of only two functions each (**xtin** and **xtout** in line 10, and **sxtin** and **sxtout** in line 13). These functions are customized versions of the **ttin** and **ttout** functions. The **nulldev** function is a null function that does not return a value. **nulldev** is described in Section D3X of the *BCI Driver Reference Manual*.

When the system is booted, the operating system takes the information from the *kernel* master file and creates a matrix in main memory called the *line discipline switch table*[1]. An example of the line discipline switch table is shown in Figure 7-4.

t_line	*open*	*close*	*read*	*write*	*ioctl*	*rxint*	*txint*	*modem int*
0	**ttopen**	**ttclose**	**ttread**	**ttwrite**	**ttioctl**	**ttin**	**ttout**	**nulldev**
1	**nulldev**	**nulldev**	**nulldev**	**nulldev**	**nulldev**	**xtin**	**xtout**	**nulldev**
2	**nulldev**	**nulldev**	**nulldev**	**nulldev**	**nulldev**	**sxtin**	**sxtout**	**nulldev**

Figure 7–4 Example Line Discipline Switch Table

NOTE: In the above table, *rxint* means receive interrupt, *txint* means transmit interrupt, and *modem int* means modem interrupt. **nulldev**(D3X) is an empty function.

Line Discipline Zero

Line discipline zero (*0* in Figure 7-4 or *Number 0* in Figure 7-6) is a set of functions that provide a terminal interface. Line discipline zero has the following characteristics:

- I/O processing functions are taken from the *tt1.c* source code file.
- support functions such as flushing input/output queues and canonical data processing are taken from the *tty.c* source code file.
- provides for interrupts
- the `clist` buffering scheme is used to convey characters

In addition to terminals, drivers for network protocols and line printers can be written with the line discipline zero. It is not usually necessary to write a driver to connect a new terminal to the system; rather, you can write a new *terminfo* file as explained on the **terminfo**(4) manual page. However, writing a *terminfo* file can only provide help for user-level programs that use the *terminfo* database.

Using the `clist`(D4X) and `tty`(D4X) data structures, the line discipline zero provides both buffering and processing of character data. All the information needed to perform I/O operations with a terminal is maintained in the `tty` structure.

1. "Line discipline" means communication line protocols for processing characters received from character devices. The line discipline switch table matches driver routines to base level and interrupt activities. This table is indexed by the **t_line** member of the tty structure.

The following lists the differences between TTY drivers and other character drivers:

- Drivers written in the TTY subsystem may have **start**(D2X) routines but not **init**(D2X) routines.
- The `tty` structure is initialized when the TTY driver is opened.
- In addition to the system entry-point routines, TTY drivers must have a **proc**(D2X) routine to process various device-dependent operations. The **proc** routine is not called by the `cdevsw` switch table. This routine can be called by assigning its address to the **t_proc** member of the `tty` structure.
- Drivers written in the TTY subsystem use a special set of functions which are described in Section D3X of the *BCI Driver Reference Manual*. Figure 7-5 shows driver routines and corresponding TTY functions:

Driver Routine	*TTY Function*	*Notes*
open	**ttopen**	Connects device to process
	ttinit	Establish default terminal settings
close	**ttclose**	Called indirectly through `linesw`
read	**ttread**	Called indirectly through `linesw`
write	**ttwrite**	Called indirectly through `linesw`
ioctl	**ttioctl**	Set device parameters
	ttiocom	Change device parameters
rint	**ttin**	Called indirectly through `linesw`
xint	**ttout**	Called indirectly through `linesw`

Figure 7–5 Line Discipline Functions in Driver Routines

Refer to *Line Discipline Functions Calling Sequences* in this chapter for more information on how each function is called.

The three AT&T line disciplines are shown in Figure 7-6.

Number	*Use*	*Defined in*
0	tty — Regular terminals (default)	*ttl.c, tty.c, and tty.h*
1	xt — AT&T bit-mapped graphics terminals such as the AT&T 630	*xt.c, jerq.h* and *xt.h*
2	xt dxt — **shl**(1) command	*sxt.c and sxt.h*

Figure 7−6 Standard Line Disciplines

The **.c* files are located in the source *io* directory appropriate for the computer in use. The **.h* files are in located in the */usr/include/sys* directory.

Writing Line Disciplines

Writing a new line discipline involves writing kernel functions that correspond to the appropriate slots in the `linesw` table. When a list of these functions is added to the line discipline switch table in the *kernel* master file and the system is reconfigured, the new line discipline is installed in the system.

The new line discipline should be given a short (but unique) name that is used as a header to the Line Discipline Switch Table and also as a prefix for the function names. Note that the **t_line** value assigned to your line discipline may vary by configuration.

Should an intelligent terminal controller deliver a character directly from the terminal with special character processing built-in, then drivers for such devices could be written without a line discipline.

Before writing a line discipline, consider the following alternatives:

1. If you need to change how data is interpreted by the terminal, you should use the **stty**(1) user command, or the **ioctl**(2) system call to modify the `termio` structure described in **termio**(7).
2. Most terminal definitions can be accomplished with a new *terminfo* file.
3. If you need to write a driver for a terminal, you may be able to use the existing line discipline zero functions and supply new device-dependent input and output routines.
4. If you need to establish a new set of character evaluation procedures, you can replace the **ttin** function.

The following three steps are required to write a line discipline.

1. Carefully planning your application to ensure that a line discipline really needs to be written. Writing a line discipline is a very complex task and most devices can be well-served by the default TTY line discipline functions (shown as *Number 0* in Figure 7-6).

2 Refer to the TTY manual pages in Section D3X, to descriptions of the **proc**(D2X) routine, and to the `tty`(D4X) structure described in the *BCI Driver Reference Manual.*

3 Writing the routines that you need for your application.

4 Putting the names of the routines in the *kernel* master file.

5 Ensuring that your driver **open**(D2X) routine sets **t_line** to the new value of your line discipline.

For most driver applications, you must supply the following:

- Device Dependent Input/Output — a driver must be written to accept data from a terminal and to send data to a terminal. This code is outside the scope of line disciplines.
- A **proc**(D2X) routine to handle calls to the device dependent input-output routines.

System calls such as **read**(2) or **write**(2) access the driver routines through the `cdevsw`(D4X) (character device switch table). Figure 7-7 illustrates how the `cdevsw` driver routines relate to the line discipline functions. For example, when the **open**(2) system call is executed on a TTY device, the **open** member of the `cdevsw` is accessed. This member in turn calls the driver **open**(D2X) routine which calls `linesw` **l_open**. The **ttopen** function is associated with **l_open** (by the *kernel* master file) and is then executed.

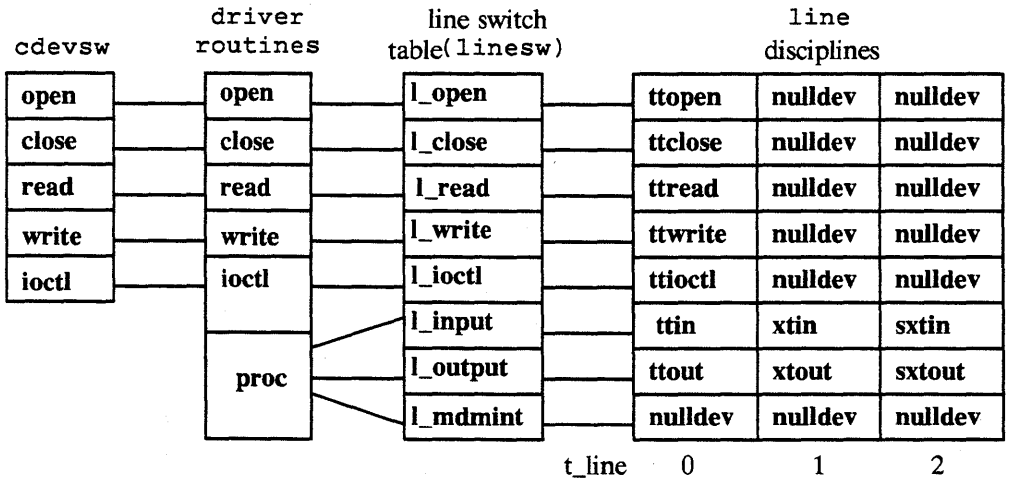


Figure 7–7 Calling Line Discipline Functions

Line Discipline Functions Calling Sequences

The following diagrams illustrate the sequence in which line discipline functions call each other and the driver **proc**(D2X) routine. The outer most box in each figure depicts the first function called. Each inner box represents a subsequent function or **proc** routine call. For example, in the first figure for **ttopen**(D3X), this function calls the **ttioctl**(D3X) function with the LDOPEN flag. The **ttioctl** function then calls the **proc** routine with the T_INPUT flag. These figures, while representative of the actual calling sequence, should not be taken as depicting all of the activities that occur within the functions or a driver routine. They are only meant to be simplified illustrations to aid in your understanding of the way these functions work.

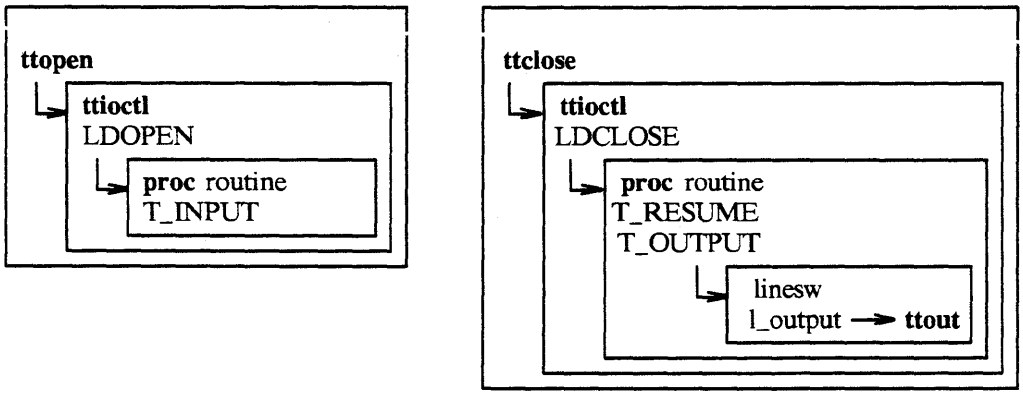


Figure 7–8 ttopen and ttclose Calling Sequence

The **ttopen** function is called from the driver **open** routine to initialize the `tty` structure. **ttopen** is called for the first terminal driver open. It calls **ttioctl** with the LDOPEN flag. **ttioctl** allocates the receive buffer and then calls the **proc**(D2X) routine with T_INPUT as the second argument. In the **proc** routine, the TTY device is prepared to receive input. This example of the **proc** routine makes no further calls to TTY functions or to itself.

The **ttclose** function is called by the driver **close** routine to release allocated resources. **ttclose** is called after the last terminal close. **ttopen** calls **ttioctl** with the LDCLOSE argument. **ttioctl** calls the **proc** routine with the T_RESUME argument. **ttioctl** then waits for the serial port UART to drain (in the **ttioctl** function), and then releases any allocated buffers. The call to the **proc** routine (T_RESUME) causes a drop-through condition to the T_OUTPUT condition which calls **ttout** through the **l_output** member of the `linesw` structure.

Calling Sequences for ttread and ttwrite

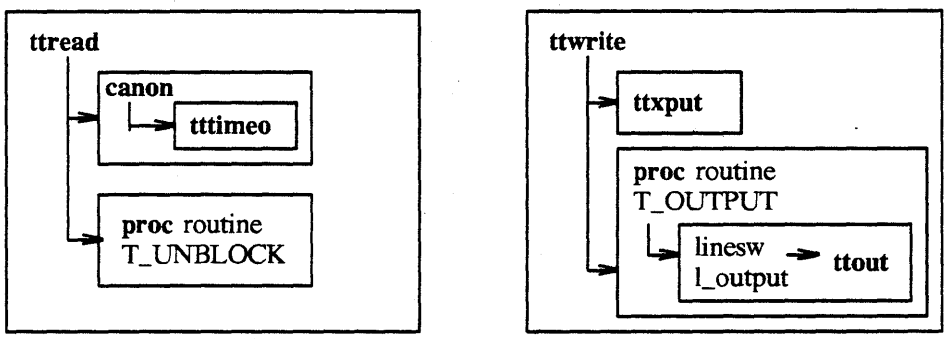


Figure 7–9 ttread and ttwrite Calling Sequence

The **ttread** function is called by the driver **read**(D3X) routine to convey input characters to the user program. **ttread** calls both the **canon**(D3X) function and the **proc** routine with the T_UNBLOCK argument. **canon** calls the **tttimeo** function (listed in this chapter).

The **ttwrite** routine is called by the driver **write** routine to convey output characters from the user program. **ttwrite** calls **ttxput** to put the characters on the TTY output queue. Then the **proc** routine is called. **proc** calls **ttout** to build up a block of characters to send to the terminal.

Calling Sequences for ttioctl and ttin

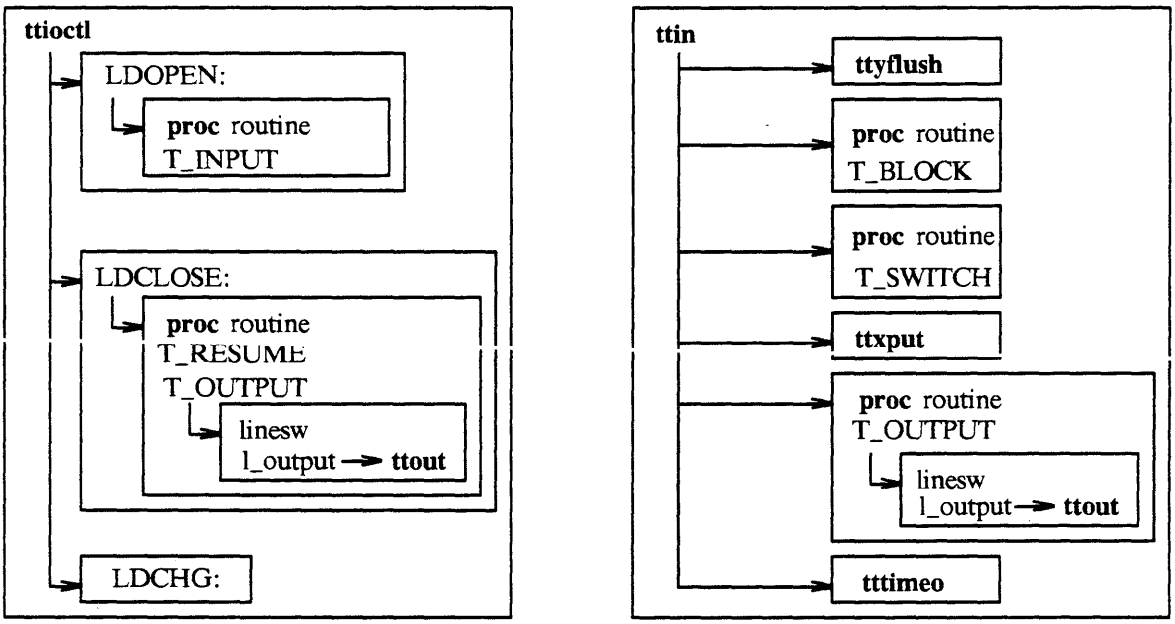


Figure 7–10 ttioctl and ttin Calling Sequence

The **ttioctl** function is called by **ttopen, ttclose**, and by **ttiocom** to set or get terminal control information. **ttioctl** has the conditions, LDOPEN, LDCLOSE, and LDCHG. In the LDOPEN condition, the **proc** routine is called. The LDCLOSE condition calls the **proc** routine. In the **proc** routine, there is typically not a break statement so control drops through to the T_OUTPUT section in the **proc**. A call is made to the **l_output** member of the `linesw` structure thus invoking **ttout**.

The **ttin** function is called from the driver interrupt routine and from **ttiocom** to process characters received from the terminal. **ttin**, depending on the condition, calls **ttyflush**. The **proc** routine is called, with T_BLOCK set and with T_OUTPUT set, which then calls **ttout** through the line switch table. The T_SWTCH condition is handled in the **sxtproc** routine (a part of the **sxt** driver for the **shl**(1) shell layers user command) which is not described in the AT&T driver interface. The T_SWTCH condition is provided for switching between context layers.

The **ttxput** command is then called. Finally, **tttimeo** is called to provide a means of timing input when VTIME (the TIME variable in **termio**(7)) is set.

Calling Sequences for ttout, ttxput, and tttimeo

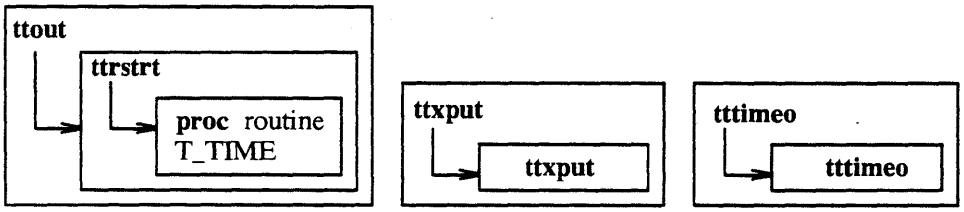


Figure 7–11 ttout, ttxput, and tttimeo Calling Sequence

The **ttout** function is called from the **proc** routine to move characters into the output queue. **ttout** calls **ttrstrt** which calls the **proc** routine for the *xt.c* driver (not covered in the AT&T driver interface). **ttout** builds a block of characters for transmission to the terminal.

The **ttxput** function is called from **ttwrite** and **ttin** to output characters to a terminal. **ttxput** calls itself when only upper case letters are being displayed.

The **tttimeo** function is called by **canon** and **ttin** to delay execution when special characters are entered to ensure that the string was entered by the user and was not entered as communications protocol. **tttimeo** calls itself after an interval determined by the value in the **termio**(7) TIME variable (in tenths of a second). **tttimeo** is listed in the *Terminal Timing Routines* section in this chapter.

Calling Sequence for ttiocom

The **ttiocom** function is called from the driver **ioctl** routine. **ttiocom** is used to flush buffers, call the line switch table **l_ioctl** member (**ttioctl**), or call the driver **proc** routine.

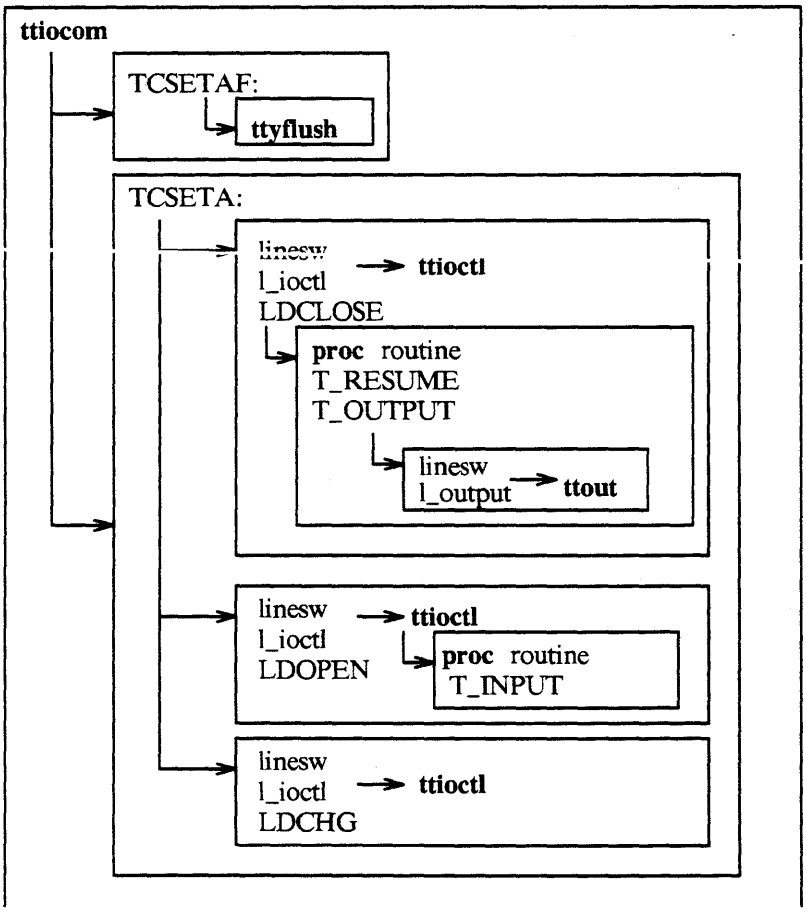


Figure 7–12 ttiocom Calling Sequence *(part 1 of 2)*

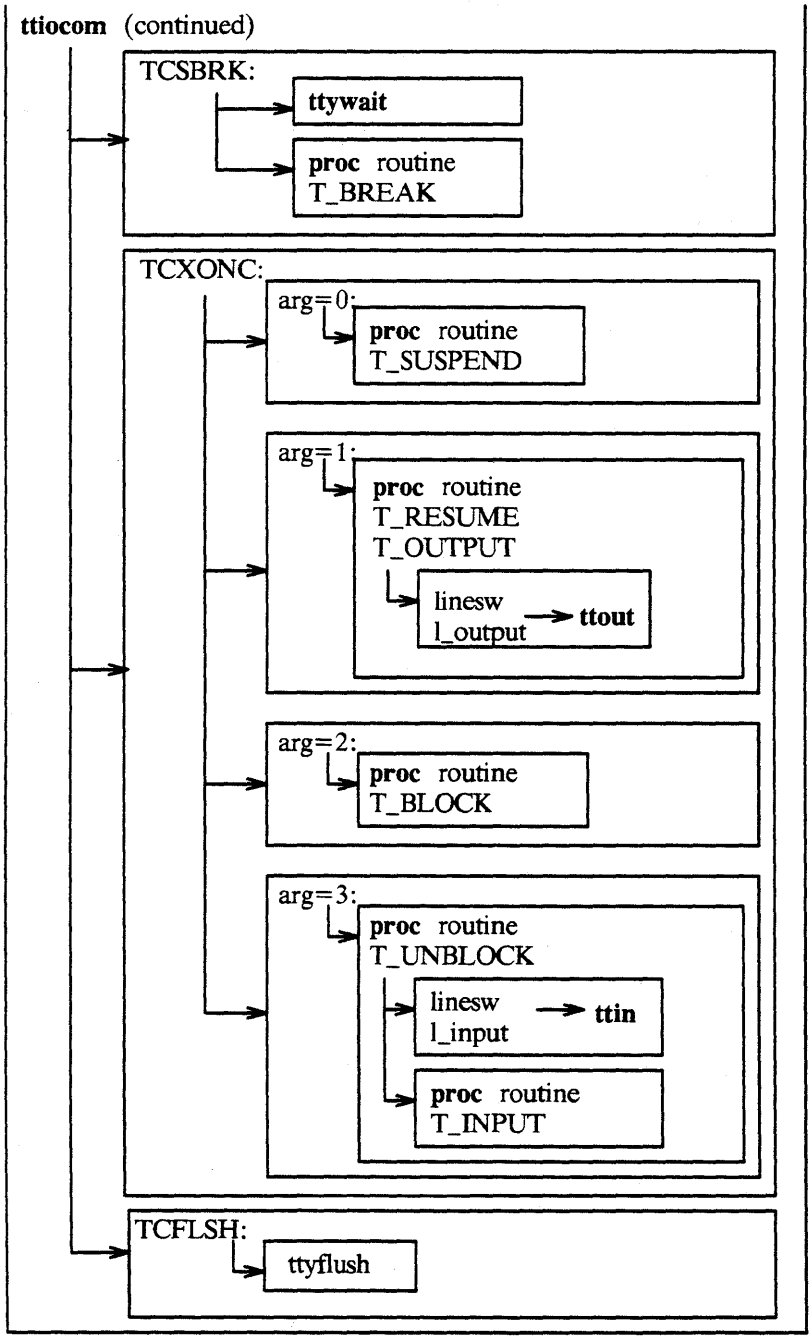


Figure 7–12 ttiocom Calling Sequence *(part 2 of 2)*

Calling Sequence for ttyflush, ttinit, ttywait, canon, and ttrstrt

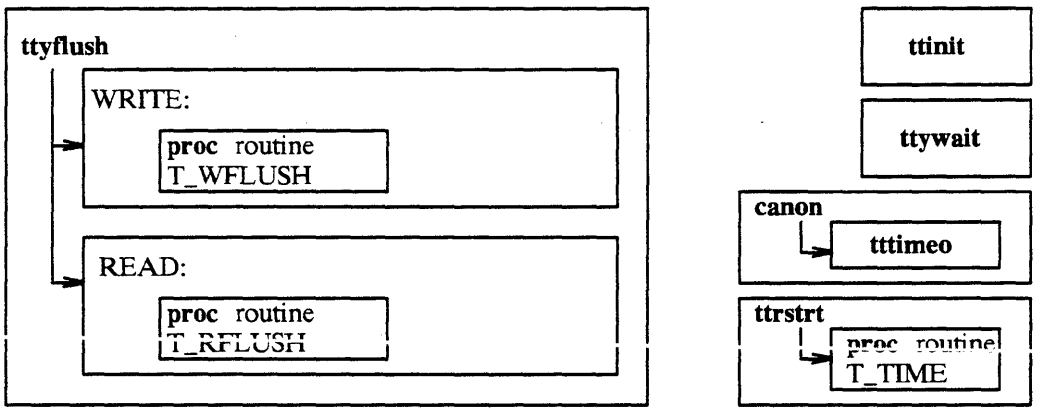


Figure 7–13 ttyflush, ttinit, ttywait, canon, and ttrstrt Calling Sequence

The **ttyflush** function is called from **ttioctl** when **ttclose** has been called, from **ttiocom**, from the driver interrupt routine, and other support routines. **ttyflush** calls the driver **proc** routine.

The **ttinit** function is called from the driver **open** routine to initialize the `tty` structure.

The **ttywait** function is called from **ttioctl, ttiocom** and from the driver **write** routine to delay process execution for 13 clock ticks to let the universal asynchronous transmitter-receiver (UART) drain. **ttywait** serves as a way of balancing timing problems that may occur between the speed of the CPU and that of the terminal.

The **canon** function is called from **ttread** to perform special processing of characters transmitted from the terminal that are outside the range of printable characters. **canon** calls **tttimeo** when handling the **termio**(7) TIME variable.

The **ttrstrt** function calls the **proc** routine with T_TIME set. T_TIME is only implemented in the **xt** driver for AT&T bit-mapped graphics terminals such as the AT&T 630.

The tty Structure

Each TTY terminal device has a `tty`(D4X) structure associated with it. The `tty` structure defines the character queues and buffers associated with the device as well as the operational modes for the device. The members of the `tty` structure can be divided into the following three groups:

1 control and status fields (**t_line, t_proc, t_pgrp, t_state, t_delct**)

2 data buffer pointers (**t_rawq, t_canq, t_outq, t_tbuf, t_rbuf**)

3 operational modes (**t_oflag, t_lflag, t_iflag, t_cflag,** and **t_cc**)

The `tty` structure manages data buffering, terminal settings, and tracks the activity of the terminal. The `termio` structure is used to retain terminal settings and functionality.

Each of the TTY functions and the **canon** function require a pointer to the current instance of the `tty` structure for the terminal you are referencing. The `tty` structure and the `termio` structure, described in **termio**(7), comprise the most important elements of the line discipline and line discipline support functions. elements of the **get*** and **put*** functions.

The line discipline functions are used to manage a series of buffers that are members of the `tty` structure. These members are

t_rawq contains the data from which the BREAK and DELETE keys have been stripped

t_canq contains the data from which the backspace and other special characters have been resolved

t_outq contains the data from the user process or echoed characters

t_tbuf contains the data ready to be transmitted

t_rbuf contains the data received from the terminal

The TTY subsystem consists of a series of buffers in which data is inserted, processed, and then extracted. The subsystem converts raw data received from a terminal into data usable by a user program. When a key is pressed on a keyboard, an interrupt is generated and **ttin**(D3X) is called from a device-dependent driver routine. **ttin** performs the following:

- conveys data from the **t_rbuf** receive buffer to the **t_rawq** raw data buffer
- echoes characters to the **t_outq** output buffer
- resolves BREAK and DELETE key entries, signaling processes if necessary

After **ttin** is called, the following functions are called to convey data between the terminal and the user program:

1. The **ttwrite** routine conveys the data from the user program to the **t_outq** output buffer.
2. The **ttout** function is called to convey the data from the **t_outq** output buffer to the **t_tbuf** transmit buffer.
3. A driver device dependent output routine sends the data to the terminal screen.

The tty and termio Structures

The `tty` structure and the `termio` structure share many similarly named members. These two structures govern the way terminals behave in the UNIX operating system. Two examples of this are how a terminal is accessed when a user logs on and how the software controls are set for a terminal. The **stty**(1) and **getty**(1M) commands are used at user level to write to the `termio` structure. These commands also call the **ttiocom** function through an **ioctl**(2) call. **ttiocom** copies the information in the `termio` structure into the `tty` structure.

This section describes the process by which the `termio` structure is populated when users log on.

The `termio` structure has a group of members that have direct counterparts in the `tty` structure. These members specify the operational modes for the device. Figure 7-14 shows how these two structures relate.

termio	*tty*	*Use*
c_iflag	**t_iflag**	input control, such as parity checking, start/stop output control, and mapping of newline to return
c_oflag	**t_oflag**	output control, such as delays on output and mapping of newline to return
c_lflag	**t_lflag**	local terminal control, such as echoing and enabling signals
c_cflag	**t_cflag**	hardware control of terminal, such as baud settings, character size, and hang up on last close
c_cc	**t_cc**	control character definitions, such as the erase and kill characters and the character to send SIGINT

Figure 7–14 Operational Modes for Terminal Devices

The fields in the `termio` structure are set by the **getty**(1M) command. **getty** is executed by the **init**(1M) command. **init** accepts as input the */etc/inittab* file which contains a line for each terminal device configured on the system. Each *inittab* terminal definition line contains a call to the **getty** command. The **getty** command sets the terminal type, its baud rate, and its associated line discipline. The driver **open** routine is called by the user level **getty** process the first time a device is opened. The **open** routine is called each time a process is spawned for a terminal subdevice.

The /etc/inittab File

The */etc/inittab* file controls processes that execute when the computer changes run level. When a new state is entered, the **init**(1M) program reads *inittab*, finds the "instructions" that apply to that run state, and executes those programs in the order in which they are listed in *inittab*. For most drivers, you will not modify *inittab* but rather create other files that will be called automatically.

Each line in *inittab* has four fields, separated by colons. A comment should be added at the end of the line; it is preceded with a "#" and can go to the end of the line.

Figure 7-15 shows the **getty**(1M) lines from a sample */etc/inittab* file. The fields are explained on the **inittab**(4) manual page.

```
1   co:234:respawn:/etc/getty console console
2   ct:234:off:/etc/getty contty contty    # Network out
3   31:234:respawn:/etc/getty tty31 9600   # Network in line #1
4   32:234:respawn:/etc/getty tty32 9600   # Network in line #2
5   33:234:respawn:/etc/getty tty33 9600   # Network in line #3
6   34:234:respawn:/etc/getty tty34 9600   # Network in line #4
7   41:234:off:/etc/getty tty41 9600    # Network out line #1
8   42:234:off:/etc/getty tty42 9600    # Network out line #2
9   43:234:off:/etc/getty tty43 9600    # Network out line #3
```

Figure 7–15 Example /etc/inittab File

The fields in the *inittab* file are:

1. **id**: One or two characters used to uniquely identify an entry.
2. **rstate**: The state or states in which this command can be executed. The valid values with their meanings are:

s,S,0,1	Single-user state
2	Multi-user state
3	Multi-user state with RFS running
4	Not currently used
5	Go to firmware mode
6	Automatic reboot

NOTE: 0 in **rstate** means power down on the 3B2 computer, but single-user on the 3B15 or 3B4000 computers. If no number is specified, the default is that the command can be executed in any run state.

More than one number can be used in this field; for instance, "56" means to execute this process when the system state switches to either state 5 or 6.

3 **action**: The conditions under which **init** should execute the process in this line. For a full explanation of all actions, see **inittab**(4) in the *UNIX System V Programmer's Reference Manual*. The options of interest to driver writers are:

wait — start process and wait for it to terminate when system first enters that runstate

bootwait — execute only once after system is booted, the first time the system enters a state that matches **rstate** for this entry.

off — do not restart this process when state changes

sysinit — used for initializing devices, identifies entries to be executed before **init** spawns a shell on the console

respawn — restart this process if it dies or if it is not already running when system state changes

4 **process**: The full pathname of the process to be invoked and arguments to the process

The /etc/gettydefs File

/etc/gettydefs defines the speed and terminal settings (IOCTL values) to be moved into the `tty` structure when the device is opened for the first time. The format of a *gettydefs* line is shown in Figure 7-16.

---

label#initial-flags#final-flags#login-prompt#next-label

For example:

```
9600#B9600#B9600 SANE IXANY TAB3 HUPCL#login:#4800
```

Figure 7−16 Format of a /etc/gettydefs Entry

The # serves as a field delimiter. The second and third fields set default I/O control command values for this device: *initial-flags* are the values assigned to this structure when it is inactive (typically only the baud rate), and *final-flags* are the values assigned when a user accesses the device, just before the **login** program executes.

If the default baud rate for the TTY port does not match the speed given in the */etc/getty* line for that device, the user can press the BREAK key, and **getty** will try a different speed, meaning a different line in *gettydefs*. The *next-label* field specifies the speed to try next.

The **getty** command can be executed without specifying the speed. In this case, the first line in *gettydefs* is the default.

The values in the third field are typically used for terminals (although the baud rate may vary). IXANY, TAB3, and HUPCL are documented on the **termio**(7) manual page. SANE is a composite flag defined in *getty.c* that sets flags to coordinate processor and terminal communication.

The I/O control commands for the `tty` structure can also be set with the **stty**(1) command in the */etc/profile* file, the user's *.profile* file, or as a user shell command. **stty** first calls the **ioctl**(2) system call. The **ioctl** system call then calls the drivers **ioctl**(D2X) routine, which in turn calls the appropriate functions from the line discipline through the `linesw` table to record the new I/O control command value in the appropriate flag or array of the `tty` structure for that terminal device.

Figure 7-19 summarizes how the operational modes in the `tty` structure are populated from `termio` values, the **getty** values associated with each `termio` member, and from **stty** commands.

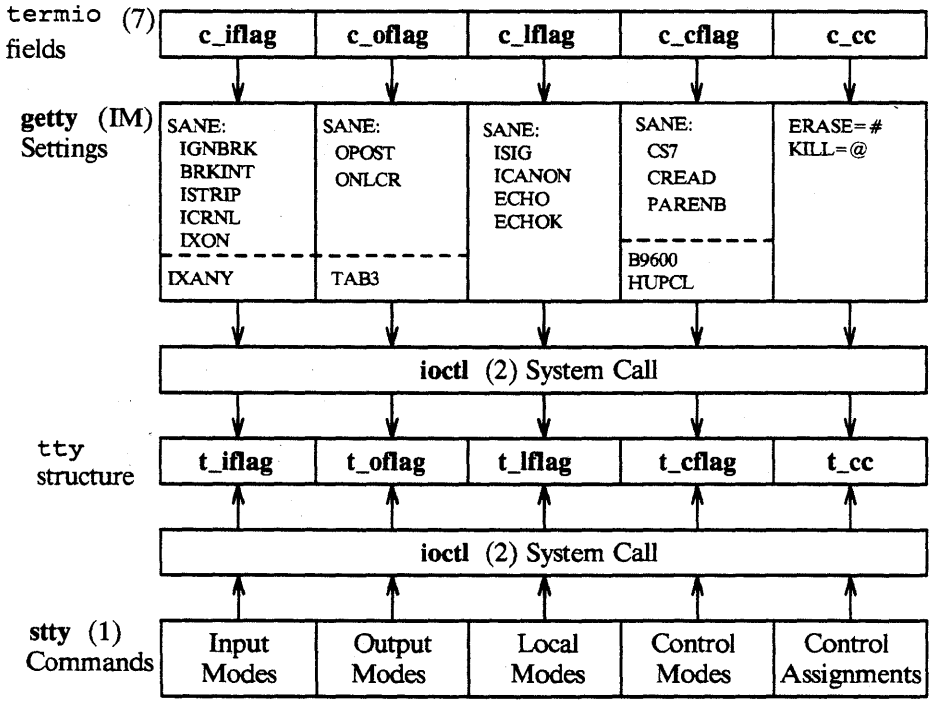


Figure 7–17 Populating the tty Operational Modes

Terminal Routines

This section describes how driver routines are constructed to take advantage of the capabilities provided in the TTY interface.

Terminal open Routines

The TTY subsystem provides two functions, **ttinit**(D3X) and **ttopen**(D3X), for the driver **open**(D2X) routine. The **ttinit** function is used only for drivers that use line discipline 0; if your driver uses its own line discipline, you must write a similar routine for that line discipline. **ttinit** performs the following:

- **t_line** is set to zero (line discipline zero)
- **t_iflag** is set to zero
- **t_oflag** is set to zero
- **t_cflag** is ORed with SSPEED (300 baud), CS8 (8-bit character size), CREAD (enable receiver), and HUPCL (hang up on last close).
- **t_lflag** is set to zero
- **bcopy**(D3X) is called to move **ttcchar** to **t_cc**. **ttcchar** is an eight-character array containing:

    1 CINTR — Delete character (octal 0177)

    2 CQUIT — Quit character (octal 034)

    3 CERASE — Erase character (#)

    4 CKILL — Kill character (@)

    5 EOF — End Of File character (CTRL-d)

    6 NULL — 0

    7 NULL — 0

    8 NULL — 0

The **ttinit** function cannot be called through the line discipline switch table, since it establishes the line discipline to be line discipline zero. If a different line discipline is used, the appropriate initialization routine should be called in place of the **ttinit** function.

The driver **open** routine (line 3 in Figure 7-18) calls the **ttinit** function (line 13) and **ttopen** via the line switch table.

When the TTY subsystem is initialized, one instance of the `tty` structure is established for each TTY port that can be configured on the system.

When a driver **open** routine is called for a terminal device, the logical state of the device is checked (line 11). If the device has not previously been opened (ISOPEN) and is not currently being opened, the `tty` structure is initialized to its default values (**ttinit** in line 13). The address to the device command processing routine is provided for the line discipline routines, and the hardware is initialized to the present baud rate and error checking settings specified in the `tty` structure.

```
1   extern struct tty   xx_tty[];   /* Location of logical device structures */
2    ...
3   xx_open(dev, flag)
4   dev_t dev;
5   {
6   register struct tty *tp;
7   register struct device *rp = &xx_addr[minor(dev) >> 3]; /* Get device regs */
8   register int    port = minor(dev) & 0x07;              /* Get port number */
9    ...
10    tp = &xx_tty[minor(dev)];
11    if ((tp->t_state & (ISOPEN | WOPEN)) == 0)  /* If device is not open and */
12    {                                          /* waiting to be opened,     */
13       ttinit(tp);       /* initialize tty structure with default values    */
14       tp->t_proc = xx_proc; /* Provide line discipline routines access to */
15                          /* the driver command processing routine    */
16       /*  The appropriate device registers would be set to match the     */
17       /*  values stored in the tty structure - hardware dependent.       */
18     } /* endif */
19  ...
```

Figure 7–18 Initializing tty Structure Default Values

The **ttopen** function establishes the connection between the process group (**t_pgrp**) and the device. It also allocates and initializes a `cblock`(D4X) for the receive buffer (**t_rbuf**) of the `tty` structure. To take care of any initialization peculiar to the device hardware, **ttopen** calls the driver **proc**(D2X) routine with the T_INPUT argument.

In Figure 7-19, when a terminal device is being opened, the driver **open** routine is responsible for establishing a physical and logical data connection. After the default settings are made in the `tty` structure, and the device registers have been set by the **ttinit** function, the driver determines if a physical connection has been made by testing carrier from the modem (line 2). If a carrier is present, the `tty` structure indicates a physical connection has been made (line 4). Otherwise, the `tty` structure indicates a physical connection has not been made (line 6).

If the process wishes to wait for carrier (line 8), and carrier is not present, the driver waits for carrier (**sleep**(D3X) in line 12). The last driver operation **open** routine is used to establish a logical data connection and associate the device to a process by making the appropriate settings in the `tty` structure (**ttopen**). In order to allow other protocols, a driver must access the **ttopen** routine through the line discipline switch table (line 15) (**l_open** is defined in *conf.h*). The **t_line** member of the `tty` structure contains the line discipline (in this case zero) and serves as the index to the line discipline switch table.

Interrupts are disabled during the **ttopen** call to ensure all parameter settings in the `tty` structure are made before any testing and resetting of the parameters is done by a driver interrupt and/or polling routines.

Refer to the **ttopen**(D3X) manual page for more information on this figure.

```
1    oldlevel = spl6();
2    if ((rp->modem_status & (0x0100 << port)) != 0) /* If there is carrier */
3    {                                    /* to the modem,       */
4       tp->t_state |= CARR_ON;          /* indicate carrier is established */
5    } else {
6       tp->t_state &= ~CARR_ON;         /* else indicate carrier is dropped */
7    } /* endif */

8    if ((flag & FNDELAY) == 0) {     /* If process wants to wait for carrier */
9       while((tp->t_state & CARR_ON) == 0) /* while carrier is not present, */
10      {                                /* indicate process is waiting   */
11          tp->t_state |= WOPEN;                    /* for carrier      */
12          sleep((caddr_t)&tp->t_canq, TTIPRI);     /* Wait for carrier */
13      } /* endwhile */
14   } /* endif */
15   (*linesw[tp->t_line].l_open)(tp);   /* Establish logical data connection */
16   splx(oldlevel);
```

Figure 7–19 Opening a tty Device

Terminal close Routines

The line discipline close function, **ttclose**, is called by the device driver **close**(D2X) routine. The **ttclose** function disassociates the device from the process that opened it and resets the ISOPEN flag in the device internal state register (**t_state**). **ttclose** calls the driver **proc** routine (with the T_RESUME argument) to transmit any characters in the device transmit buffer (**t_tbuf**) out to the terminal, clears out all the TTY buffers and queues, and returns all `cblock`(s) allocated to the device.

On the last close of a terminal device, the driver **close**(D2X) routine (line 6 in Figure 7-20) terminates the logical data connection and disassociates the device from a process that is specified in the `tty` structure (**ttclose**). In order to allow other protocols, a driver must access the `ttclose` function through the line discipline switch table (**l_close** is defined in *conf.h*).

After the logical data connection is terminated, the driver would break the physical connection (such as instructing the modem to drop carrier) (line 6).

```
1  extern struct tty xx_tty[];   /* Location of logical device structure  */

2  xx_close(dev)
3  dev_t dev;
4  {
5  register struct tty *tp = xx_tty[minor(dev)]; /* Get device tty structure */

6       (*linesw[tp->t_line].l_close)(tp);  /* Break logical data connection */
7       ...
```

Figure 7–20 Data Connection is Terminated

Terminal read Routines

When a process requests data from a terminal device, the driver **read**(D2X) routine locates the `tty` structure associated with the device. The character data is copied from the input queues to the user data area using **ttread**.

ttread calls **canon** to perform canonical processing of data (erase, kill, and escape) as it transfers characters from the raw queue to the canonical queue. If no characters are available, it calls **sleep** to wait on the address of the raw queue until characters become available. After canonical processing, **ttread** transfers data from the canonical queue to user data space. If transmission from the terminal is blocked because the number of characters in the raw input queue is above the high water mark, and if the read causes that number to go below a safe level, **ttread** calls the driver **proc** routine (with the T_UNBLOCK argument) to resume transmission from the terminal. To allow for alternative line protocols, a driver must access the **ttread** function through the line discipline switch table (line 7 in Figure 7-21). **ttread** is accessed through the **l_read** member of the `linesw` table which is defined in *conf.h*.

```
1  extern struct tty    xx_tty[];   /* Location of logical device structures */
2   ...
3  xx_read(dev)
4  dev_t dev;
5  {
6       register struct tty *tp = &xx_tty[minor(dev)];

7       (*linesw[tp->t_line].l_read)(tp); /* Copy character data from input */
8                                         /* queues to user data area      */
9  } /* end xx_read */
```

Figure 7–21 Processing an Input TTY Character

Terminal write Routines

Displaying a character on the screen of a terminal is simpler than reading information from the keyboard since only one queue, the output queue (**t_outq**), is involved. Still, activities at both base and interrupt levels are involved. A transmit buffer provides the buffering of characters between the base and interrupt portions.

The terminal driver **write**(D2X) routine calls **ttwrite** to move the characters output from the user data space to the output queue. **ttwrite** calls the driver **proc** routine with T_OUTPUT set to get **ttout** to transmit the data to the terminal.

Once initiated, output is sustained by interrupts from the device. A transmit-complete interrupt causes control to be passed to the driver transmit interrupt handler. The driver outputs the next character in the transmit buffer to the device. If the output buffer is empty, **ttout**(D3X) is called to move characters from the output queue to the buffer.

The driver **write** routine receives the device number as an argument. It uses this argument to determine the `tty` structure for the device being written. This is then passed to **ttwrite**.

The **ttwrite** function transfers characters from user data space to the output queue as long as the output queue high water mark has not been exceeded. The characters are processed as they are put on the output queue to expand tabs and to add appropriate delays for newline, carriage return, and backspace characters. When the high water mark is reached, **ttwrite** calls **sleep** to wait on the output queue.

When a process requests data be transferred to a terminal device, the driver **write** routine locates the `tty` structure associated with the device (line 3 in Figure 7-22). The data is copied from the user data area to the output queues with **ttwrite** (line 7). **ttwrite** is accessed through the **l_write** member of the `linesw` table which is defined in *conf.h*.

```
1  extern struct tty    xx_tty[];    /* Location of logical device structures */
2   ...
3  xx_write(dev)
4  dev_t dev;
5  {
6       register struct tty *tp = &xx_tty[minor(dev)];

7       (*linesw[tp->t_line].l_write)(tp); /* Copy character data from user */
8                                          /* data area to output queues    */
9  } /* end xx_write */
```

Figure 7–22 The ttwrite Function

Terminal ioctl Routines

Changing the many parameters associated with terminal devices requires close cooperation between the driver and the TTY subsystem. The **ttiocom** function provides access to reading and changing the various TTY parameters contained in the `tty` structure. Changing such parameters usually requires that device registers also be altered. The driver is responsible for changing these registers.

A request to read or change terminal parameters is initiated by an **ioctl**(2) system call from a user process. This causes the driver **ioctl**(D2X) routine to be called. The driver locates the `tty` structure associated with the device and calls the common **ioctl** routine, **ttiocom**.

Internally, **ttiocom** calls **ttioctl**(D3X). These two functions together affect the appropriate parameter settings and return to the driver. Although **ttiocom** and **ttioctl** are together involved in parameter access, each has a different purpose. **ttiocom** is a general-purpose function providing common parameter handling. **ttioctl** is specialized in that it deals with parameters related to buffering and character processing and is associated with the terminal protocol or line discipline.

A user process can get or set terminal parameters with the **ioctl**(2) system call. All standard **termio**(7) commands access parameters in one or more of the members in the `tty` structure, and possible changes to these parameters are made first (**ttiocom**). If changes are made in the parameters of the `tty` structure, then the device registers may also need to be altered; the driver would make the necessary changes upon return from the **ttiocom** function.

NOTE: Do not call the **ttioctl** function directly. This function should always be called through the line discipline.

```
1   extern struct device xx_addr[];  /* Location of physical device registers */
2   extern struct tty    xx_tty[];   /* Location of logical device structures */
3    ...
4   xx_ioctl(dev, cmd, arg, flag)
5   dev_t  dev;
6   caddr_t arg;
7   {
8      switch(cmd)
9      {
10         /* Driver specific commands would be handled by the case */
11         /* statements, such as getting the device registers.     */

12     default:        /* Handle termio(7) commands; if invalid command is  */
13                    /* present ttiocom will update u.u_error with EINVAL */
14     {
15     register struct tty *tp = &xx_tty[minor(dev)];  /* Get tty structure */

16     if  (ttiocom(tp, cmd, arg, flag) == 1)  /* Get or set tty parameters; */
17     {                        /* If tty parameters are changed, then    */
18                              /* change the necessary device registers. */
19           register struct device *rp;
20           rp = &xx_addr[minor(dev) >> 3]; /* Get device regs */
21       /*  The changes are usually determined by examining the parameter   */
22       /*  settings in the t_iflag, t_oflag, t_cflag, and t_lflag members  */
23       /*  of the tty structure for changes like baud rate, type of parity */
24       /*  testing, etc.  --  hardware dependent.                          */
25      } /* endif */

26     } /* endswitch */
27  } /* end xx_ioctl */
```

Figure 7–23 Changing Device Parameters

Terminal Interrupt Routines

Interrupts can be handled by a single **int**(D2X) routine or with the **rint**(D2X)/**xint**(D2X) routine pair.

After a driver **rint** (receive interrupt) routine validates an input character, it stores the character in the receive buffer (**t_rbuf**). When the receive buffer is filled, the receive buffer is added to the raw queue and a new receive buffer is allocated (**ttin**). In order to allow other protocols, a driver must access the **ttin** routine through the line discipline switch table, `linesw`. The **t_line** member of the `tty` structure contains the line discipline number and serves as the index to the line discipline switch table.

If the number of characters in the raw queue exceeds a level called the high water mark, **ttin** calls the driver **proc**(D2X) routine to send a stop character to the device. When the raw queue character count exceeds the TTYHOG level of 256 characters, **ttin** flushes the `tty` structure input queues. TTYHOG is defined in the *tty.h* header file. If the interrupt character (SIGINT), typically DEL or the quit character (SIGQUIT), is found, **ttin** sends the appropriate signal to the process group associated with the device. If processes associated with the device are executing **sleep**(D3X) and **ttin** finds a line delimiter character, **ttin** awakens the process that called **sleep**.

The **ttin** function can also transmit characters to the terminal for display by calling **ttxput**.

When the terminal operates in raw mode, the fifth and sixth elements of the `tty` structure control character array indicate the number of characters needed (VMIN), and the amount of time waited before processes associated with the device should be awakened (VTIME). If the minimum character count has been met (**t_delct**), **ttin** awakens processes associated with the terminal.

```
1   struct  device              /* Layout of physical device registers */
2   {
3         int     control;    /* Physical device control word       */
4         int     status;     /* Physical device status word        */
5         short   recv_char;  /* Receive character from device      */
6         short   xmit_char;  /* Transmit character to device       */
7   }; /* End device */

8   extern struct tty   xx_tty[]; /* Location of logical device structure  */
9.  extern struct device xx_addr[]; /* Location of physical device registers */
10  extern int          xx_cnt;   /* Number of physical devices          */
11   ...
12  xx_rint(board)
13  int board;      /* The hardware board that caused the interrupt */
14  {
15  register struct device *rp = xx_addr[board]; /* Get device registers */
16  register struct tty *tp;
17  register int c, port;

18  while((c = rp->recv_char) & DATAVALID) != 0) /* While there is valid data */
19  {                           /* in the input register, retrieve it */
20     port = (c >> 8) & 0x7;           /* Get terminal's port number     */
21     tp = &xx_tty[(board << 3) & port]; /* Get corresponding tty structure */

22  /*   After the character has been checked for errors and is stripped to   */
23  /*   proper bit size, the character is stored in the receive buffer.      */

24     *tp->t_rbuf.c_ptr++ = c;     /* Store input character in receive buffer */
25     if (--tp->t_rbuf.c_count == 0)   /* If the receive buffer is full,     */
26     {      /* reset the c_ptr to first character in the receive buffer. The */
27            /* driver must do this operation to insure the buffer is added   */
28        tp->t_rbuf.c_ptr -= tp->t_rbuf.c_size; /* to the raw queue correctly */

29        (*linesw[tp->t_line].l_input)(tp); /* Add receive buffer to raw;     */
30                                    /* queue; get empty receive buffer*/
31     } /* endif */
32  } /* endwhile */
33   ...
```

Figure 7–24 ttin — Move Character to Raw Queue

The **ttout** function is called by the driver transmit interrupt (**xint**(D2X)) routine. **ttout** is passed the address of the `tty` structure associated with the device.

The **ttout** function moves characters from the output queue to the transmit buffer in preparation for output by the driver. The **ttout** function implements the actual timing delays needed during output. When it detects a delay in the output queue, it uses the **timeout**(D3X) function to arrange for an entry after the appropriate time has elapsed. This delayed entry invokes the driver **proc**(D2X) routine to resume output (from **ttrstrt**). The **ttout** function also awakens the sleeping processes when a sufficient number of characters have been transmitted; that is, when the number of characters in the output queue is less than the low water mark.

A driver transmit routine is entered when a device is ready to receive data. While the device is ready to receive data and the transmit register is free, a character is taken from the transmit buffer (**t_tbuf**) and placed in the transmit register. The state of the `tty` structure is changed to show a character is present in the transmit register and the driver command process routine is called to complete the output.

The command processing routine determines the output port. If output is blocked or there is no output for that port, then return to the caller. When the transmit buffer (**t_tbuf**) is empty, the buffer is returned to the free list and a new transmit buffer is allocated from the output queue (**ttout**). The output character is transmitted to the device and the state of the `tty` structure is changed to show the transmit register is empty.

```
1   struct  device              /* Layout of physical device registers */
2   {
3         int     control;    /* Physical device control word       */
4         int     status;     /* Physical device status word        */
5         short   modem_status;/* Modem carrier (upper 8 bits) & ring */
6                             /* (lower 8 bits) status word         */
7         short   recv_char;  /* Receive character from device      */
8         short   xmit_char;  /* Transmit character to device       */
9   }; /* End device */

10  extern struct device xx_addr[];  /* Location of physical device registers */
11  extern struct tty    xx_tty[];   /* Location of logical device structures */
12   ...
13  xx_xint(board)
14  int board;      /* Board that caused the interrupt */
15  {
16  register struct tty *tp;
17  register struct device *rp = &xx_addr[board]; /* Get device regs */
18  register struct ccblock *cp;
19  register int    port;
```

Figure 7–25 A Driver Accesses ttout Function *(part 1 of 3)*

```
20  port = rp->status & 0x7;              /* Get terminal's port number      */
21  tp   = &xx_tty[(board << 3) & port];  /* Get corresponding tty structure */
22  cp   = &tp->t_tbuf;                   /* Get transmit buffer             */

23  while((rp->status & XX_TXRDY) != 0) /* While the device is ready for      */
24  {                                   /* a character to be transmitted      */
25     if (tp->t_state & BUSY)          /* If xmit_char register is clear     */
26     {                                /* and there is more data to send,    */
27        if (cp->c_count > 0)          /* If there is data in the tbuf of the */
28        {                             /* tty structure, then give device the */
29           rp->xmit_char = *cp->c_ptr++; /* next character for transmission */
30           cp->c_count--;             /* update counter of the number of    */
31                                      /* characters remaining for output    */
32        } /* endif */
33        tp->t_state &= ~BUSY;     /* Indicate xmit_char register is primed   */
34        xx_proc(tp, T_OUTPUT);    /* test to see if output is blocked and if */
35                                  /* not enable controller for transmission  */
36     } else {
37        rp->control |= XX_TXDONE;     /* Indicate all data for port has been */
38        break;                        /* transmitted; terminate loop         */
39     } /* endif */
40  } /* endwhile */

41  } /* end xx_xint */

42  xx_proc(tp, cmd)          /* Driver command processing routine */
43  register struct tty *tp;
44  int cmd;
45  {
46  register int    dev = tp - xx_tty;          /* Compute minor device number */
47  register struct device *rp = &xx_addr[dev >> 3];      /* Get device regs */
48  register int    portmask = 0x0100 << (dev & 0x7); /* Setup output port mask */

49  switch(cmd)
50  {
51    ...
52  case T_OUTPUT:        /* Perform output processing of data to the device */
53  resume_output:
```

Figure 7–25 A Driver Accesses ttout Function *(part 2 of 3)*

```
54  {
55      register struct ccblock *cp = &tp->t_tbuf;

56      if ((tp->t_state & (BUSY | TTSTOP)) != 0)    /* If there is no data to  */
57          break;    /* transmit or output is blocked by a CTRL-s, do nothing */

58      rp->xmit_char |= portmask;  /* Enable controller to transmit character  */

59      if (cp->c_ptr == NULL || cp->c_count == 0)  /* If there is no tbuf or  */
60      {                           /* the tbuf is empty, then get a new one */
61          if ((*linesw[tp->t_line].l_output)(tp) & CPRES) == 0)  /* If there  */
62              break;          /* is no more output data, then terminate output */
63      } /* endif */

64      tp->t_state |= BUSY;  /* Indicate there is more output data in the tbuf */
65                            /* and that the xmit_char register is clear       */
66      break;
67  }  /* end T_OUTPUT case */
68  ...
```

Figure 7–25 A Driver Accesses ttout Function *(part 3 of 3)*

Terminal proc Routines

The **proc**(D2X) routine processes information received from and sent to a TTY device. The **proc** routine is unique in that it is called from both kernel TTY functions and other driver routines (including itself). If you are using the **ttiocom, ttioctl, ttin, ttread, ttrstrt, ttwrite,** or **ttyflush** functions in your driver, you must have a **proc** routine. The format for a **proc** routine is similar to that of an **ioctl** routine in that the contents of the **proc** routine are little more than a series of conditions that evaluate the *cmd* argument passed into the **proc** routine.

Figure 7-26 lists the case conditions that must be included in a **proc** routine (if the TTY function is used). See the *BCI Driver Reference Manual*, section D2X, for explanations of the case conditions provided in this table.

Case	*Required By*	*Condition*
T_BLOCK	**ttin**	if `(tp->t_rawq.c_cc>TTXOHI)` and `(tp->t_iflag&IXOFF) && !(tp->t_state&TBLOCK)`
T_BLOCK	**ttiocom**	When **ttiocom** *cmd* = TCXONC and *arg* = 2
T_BREAK	**ttiocom**	When **ttiocom** *cmd* = TCSBRK and *arg* = 0
T_INPUT	**ttioctl**	When **ttioctl** *cmd* = LDOPEN and if `tp->t_rbuf.c_ptr == NULL`
T_OUTPUT	**ttin**	if `tp->t_iflag&ECHO`
T_OUTPUT	**ttwrite**	When ready to send character to terminal
T_RESUME	**ttiocom**	When **ttiocom** *cmd* = TCXONC and *arg* = 1
T_RESUME	**ttioctl**	When **ttioctl** *cmd* = LDCLOSE
T_RFLUSH	**ttyflush**	When flushing read buffers
T_SUSPEND	**ttiocom**	When **ttiocom** *cmd* = TCXONC and *arg* = 0
T_SWTCH	**ttin**	if (tp->t_iflag&ISIG), the next character in `tp->t_cc` is VSWTCH, and if not `tp->t_iflag&NOFLSH` (**sxt** driver only)
T_TIME	**ttrstrt**	Whenever function is called (**xt** driver only)
T_UNBLOCK	**ttiocom**	When **ttiocom** *cmd* = TCXONC and *arg* = 3
T_UNBLOCK	**ttread**	If `tp->t_state&TBLOCK` and `tp->t_rawq.c_cc<TTXOLO`
T_WFLUSH	**ttyflush**	When flushing write buffers

Figure 7–26 proc Routine case Statements

Terminal Timing Routines

Occasionally, a terminal driver must provide a timing routine to wait for buffers, for a character to be entered, or to cushion differences in baud rates between the terminal and the CPU. The **ttrstrt** and **tttimeo** functions are used for theses purposes. In addition, the **delay, sleep, timeout**, and **untimeout** functions described in Chapter 9 provide additional timing capability.

The **ttrstrt** function restarts TTY output following a delay timeout. The name of the function to be executed is assigned to **tp->t_proc** before calling **ttrstrt**.

When a TCSBRK command is issued in a **ioctl**(2) system call, the line discipline routine **ttiocom** calls the driver **proc** routine with the T_BREAK argument. The purpose of the driver **proc** routine is to send a break to the device. After the break is sent, output must be suspended for 250 milliseconds. The **timeout**(D3X) function is used to call **ttrstrt** after the 250 milliseconds have elapsed. The **ttrstrt** function will call the driver command processing routine with the T_TIME command so that output can be resumed.

```
1   case T_BREAK:                   /* Send a BREAK to a device       */
2       rp->control |= XX_BRK;      /* Enable a break to be sent      */
3       rp->xmit_char |= portmask;  /* Enable controller/specify port */
4       tp->t_state |= TIMEOUT;     /* Timeout condition in progress  */
5       timeout(ttrstrt, tp, HZ/4); /* Disable timeout in 1/4 of a    */
6                                   /* second (HZ)-250 milliseconds   */
7       break;
8   ...
```

Figure 7–27 Restart TTY Output After a Delay

The **tttimeo** function is normally used in conjunction with the **canon** function's VTIME option, which is the same as the **termio**(7) TIME variable. However, **ttimeo** can be used independently to time events. Figure 7-28 gives the code for **tttimeo**:

```
1   tttimeo(tp)
2   register struct tty *tp;
3   {
4       tp->t_state &= ~TACT;
5       if (tp->t_lflag&ICANON || tp->t_cc[VTIME] == 0)
6           return;
7       if (tp->t_rawq.c_cc == 0 && tp->t_cc[VMIN])
8           return;
9       if (tp->t_state&RTO) {
10          tp->t_delct = 1;
11          if (tp->t_state&IASLP) {
12              tp->t_state &= ~IASLP;
13              wakeup((caddr_t)&tp->t_rawq);
14          }
15      } else {
16          tp->t_state |= RTO|TACT;
17          timeout(tttimeo, tp, tp->t_cc[VTIME]*(HZ/10));
18      }
19  }
```

Figure 7–28 tttimeo Function

Using the clist Buffering Scheme

A `clist` structure is the head of a linked list queue of `cblocks` that have been assigned to the driver. It contains a total count of the characters in the queue with pointers to the first and last `cblocks` in the queue.

The `clist` buffering scheme buffers small amounts of data using a `clist` or `cblock` (character list or character block). Interactive devices, such as terminals, use the `clist` buffering scheme through the TTY line discipline routines which manage the structures and I/O transfers. Terminal drivers do not need to use the `clist` buffering scheme; the driver writer is free to implement any type of data buffering scheme needed (including none) in a terminal driver.

Each `cblock` contains arrays in which the actual characters are stored, as well as indices for the first (`c_first`) and last (`c_last`) valid characters in the array. Each `c_block` contains 64 characters.

The `cfreelist` structure is the system pool of available `cblocks`, and is shared by all TTY devices on the system. The `chead` data structure heads it, and contains a pointer to the next available `cblock`, the size of the `cblock` structure, and a flag that indicates when a process is waiting for a `cblock`.

The `chead` and `cfreelist` structures should never be accessed directly, but only through the `clist` routines.

Figure 7-29 illustrates the `clist` buffering scheme.

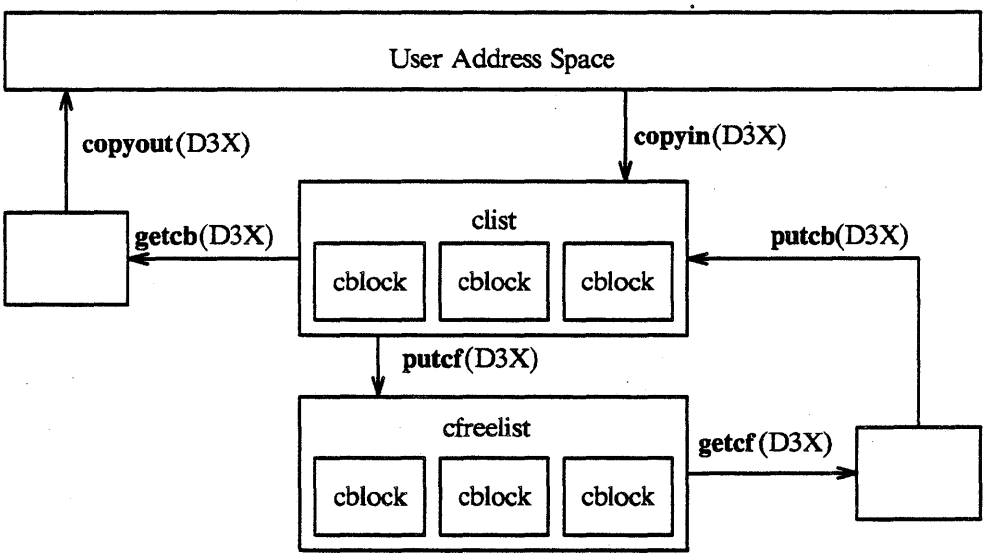


Figure 7–29 clist Buffering Scheme

To use the `clist` buffering scheme, the driver code must include the header file *tty.h*. The following table describes the functions used to read and write character lists. Each of these has a corresponding reference page in Section D3X of the *BCI Driver Reference Manual*. Note that the **copyin**(D3X) and **copyout**(D3X) functions are only described here as functions that are useful when writing character handling routines. Refer to Chapter 6 for more information on these two functions.

Function	*Activity*
copyin	copy data from user address space to driver buffer
copyout	copy data from driver buffer to user address space
getc	get a character from the `clist`
getcb	get first `cblock` on a `clist`
getcf	get a free `cblock` from system `cfreelist`
putc	put character at end of `clist`
putcb	link a `cblock` to the end of `clist`
putcf	return `cblock` to `cfreelist`

Figure 7–30 Functions for Manipulating clist Buffers

Chapter 8: Input/Output Control (ioctl)

Contents

Introduction

The **ioctl**(D2X) routine provides character-access drivers with an alternate entry point that can be used for almost any operation other than a simple transfer of characters in and out of buffers. Most often, an I/O control command is used to control device hardware parameters and establish the protocol used by the driver for processing data.

After the user-level program opens a special device file, it can pass I/O control command arguments. The kernel looks up the device's file table entry, determines that this is a character device, and looks up the entry point routines in `cdevsw`. The kernel then packages the user request and arguments as integers and passes them to the driver's **ioctl** routine with the **copyin**(D3X) or **copyout**(D3X) function. The kernel itself does no processing of a I/O control command, so it is up to a user program and a driver to agree on what the arguments mean.

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

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

Because the kernel does not interpret a command that defines an operation, a driver is free to define its own commands.

Drivers that use an **ioctl** routine typically have a command to read the current I/O control command settings, and at least one other command that sets new settings. You can use the mode argument to determine if the device unit was opened for reading or writing, if necessary, by checking the FREAD or FWRITE setting.

The **ioctl** routine can be used for transferring large chunks of data, such as when you need to pump (download) data into the driver itself (not through the driver to the hardware). In this case, the operation argument is a pointer to a buffer of an appropriate size that contains the data. The buffer itself should be set up by a user-level process or daemon.

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

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

It is critical that I/O control command definitions and routines be commented thoroughly. Because there is so much flexibility in how I/O control commands are used, uncommented I/O control commands are very difficult to interpret at a later time.

Defining I/O Control Command Names and Values

The I/O control command name is passed as the second argument (*cmd*) to the driver **ioctl** routine. It should be defined, along with an integer value that is actually passed, in the header file.

The I/O control command name and value can be defined in the driver code itself, but this is not recommended. If I/O control commands are defined in a header file, the user program and the driver can both access the same definitions to ensure that they agree about what each I/O control command value represents.

The I/O control command name is traditionally an all uppercase alphabetic string. This alphabetic name can be a mnemonic. You should try to keep the values for your I/O control commands distinct from others on the system. Each driver's I/O control commands are discrete, but it is possible for user-level code to access a driver with an I/O control command that is intended for another driver, which can lead to serious consequences, such as if it meant to pass "drop carrier on a communication line," but instead sends the argument to a disk where it is interpreted as "reformat drive." Permissions can be set to prevent most such events, but the more unique your I/O control command values are, the safer you are. Each driver has up to 2^{32} values that can be passed as an integer, so it is quite possible to avoid using numbers that are already in use.

A number of different schemes are legal for assigning values to I/O control command names. The most straightforward is to use decimals; for example

```
#define COMMAND1    01
#define COMMAND2    02
```

Similarly, one can assign hexadecimal numbers as values

```
#define COMMANDA     0x0a
#define COMMANDFF    0xff
```

The drawback to these methods is that one quickly gets an operating system that contains several instances of each I/O control command value, with the inherent risks discussed above.

A common method to assign I/O control command values that are less apt to be duplicated is to use a left-shifted 8 scheme. For instance

```
#define COMMAND10    ('Q'<<8|10)
#define COMMAND11    ('Q'<<8|11)
#define COMMAND12    ('Q'<<8|12)
```

Alternately, the shift-left-8 scheme can be defined as a constant then used for the I/O control command definitions. For example

```
#define ROTA          ('q'<<8)
#define COMMAND23     (ROTA|234)
#define COMMAND25     (ROTA|254)
```

An alternative coding style is to use enumerations for the command argument, to allow the compiler to do additional type checking

```
typedef enum    {
                  XX_COMMAND10 = 'Q'<<8 | 10,
                  XX_COMMAND11 = 'Q'<<8 | 11,
                  XX_COMMAND12 = 'Q'<<8 | 12,
} xx_cmds_t; ;
```

Coding the ioctl Routine

The format for an **ioctl**(D2X) is

```
prefixioctl(dev, cmd, arg, mode)
dev_t dev;
int cmd, arg, mode;
```

The arguments are

dev a device number (both the major and minor number)

cmd the type of operation ("command")

arg an optional argument to the operation (often specifying the address of the structure in the user program that contains settings for the hardware)

mode an optional argument containing values set when the device was open

The **ioctl** routine is coded with instructions on the proper action to take for each I/O control command. Generally, a driver **ioctl** routine consists of a **case** statement for each I/O control command that identifies the required action. The command passed to a driver by a user process is an integer value that is associated with an I/O control command name in the header file.

The **case** statement should have a "default" case to send an error value if the driver is called with an unknown I/O control command.

The general shape of an **ioctl** routine is illustrated in Figure 8-1. Note that the I/O control command definitions are shown as part of the driver code in this example, although in practice these should be defined in the header file.

For a full example of an **ioctl** routine, see the driver in Appendix E, "Sample Block Driver."

```
1       #define COMMAND1 01
2       #define COMMAND2 02
3       #define COMMAND3 04
4       extern int SUBDEVICES;

5       struct send_to_device
6       {
7       int flags;
8       char setup[64];
9       };
```

Figure 8–1 Sample ioctl Routine, *part 1 of 2*

```
10  struct receive_from_device
11      {
12      int flags;
13      char current_status[64];
14      };

15      xxioctl( dev, cmd, val,flag)
16      int dev;
17      int cmd;
18      caddr_t val;
19      int flag;
20      {

21              switch(cmd)
22              {
23              case COMMAND1:
24              /* send new status setup to device */
25                      senddev((struct send_to_device *) val));
26                      return;

27              case COMMAND2:
28              /* get current status from device */
29                      recdev((struct receive_from_device *) val));
30                      return;

31              case COMMAND3:
32              /* return number of devices */
33                      *val = SUBDEVICES;

34              default:
35                      u.uerror = EIO;
36                      break;

37              }
38      }
```

Figure 8–1 Sample I/O Control Command Routine, *part 2 of 2*

AT&T-Defined I/O Control Commands

The following tables show the I/O control commands that are included in any of the UNIX System V releases for the supported machines, along with the integer value of the I/O control command and the header file where it is defined.

Table 8–1 AT&T Defined I/O Control Commands

Command	Value	Header File	Description
AIC_IOC		*aic.h*	
BIOC		*extbus.h*	For BUS ioctl() commands
B_ADJ_DL	0	*btdl.h*	download to specified adjunct
B_ADJ_DUMP	2	*btdl.h*	dump specified area of adjunct physical memory
B_ADJ_EXEC	1	*btdl.h*	transfer control to specified address in adjunct boot image
B_EDSD	'B' <<8 | 3	*extbus.h*	Regenerate and return Extended DSD structure
B_GETDEV	'B' <<8 | 2	*extbus.h*	Get device for pass through
B_GETTYPE	'B' <<8 | 1	*extbus.h*	Get bus and driver name
B_REDT	'B' <<8 | 4	*extbus.h*	Read extended equipped device table (EDT)
B_WEDT	'B' <<8 | 5	*extbus.h*	Write extended EDT
CM_BLK_ALARM	0x2	*cman.h*	ABUS bulk power alarm
CM_FAN_ALARM	0x1	*cman.h*	ABUS minor fan alarm
CM_IC_FCSTATE	0xb	*cman.h*	force configuration state of an APE
CM_IC_GACT	0x5	*cman.h*	get a copy of the ACT
CM_IC_GDEV	0xa	*cman.h*	get the generic dev_t for the sdf
CM_IC_GSTOP	0x8	*cman.h*	gracefully stop an APE
CM_IC_HTEST	0x7	*cman.h*	host error handling test
CM_IC_MINOR	0x3	*cman.h*	determine if a minor alarm exists
CM_IC_PRIVPUB	0x9	*cman.h*	make an APE private or public
CM_IC_SCONF	0x6	*cman.h*	SCSI configuration change
CM_IC_START	0x1	*cman.h*	start an APE
CM_IC_STOP	0x2	*cman.h*	stop an APE
CM_SCSI_START	0x1	*cman.h*	start a SCSI device
CM_SCSI_STOP	0x2	*cman.h*	stop a SCSI device
C_ABORT	14	*msbih.h*	remove all packets for specified BIC
C_DIAG_STATUS	0	*msbih.h*	get diagnostic status
C_DL_SCN	11	*msbih.h*	download a section
C_EPOCH	17	*msbih.h*	toggle the time epoch flag
C_FGETSTATUS	1	*msbih.h*	force read of BIC's status register
C_GETMODE	2	*msbih.h*	get operational mode
C_GETOPTIONAL	3	*msbih.h*	get optional MSBI internal statistics
C_GETSTATS	4	*msbih.h*	get MSBI internal statistics
C_GETSTATUS	5	*msbih.h*	return last normal read of BIC's status register
C_INIT	6	*msbih.h*	initial internal MSBI storage
C_RBICVERS	16	*msbih.h*	RBIC version number
C_RESET	7	*msbih.h*	physically reset MSBI
C_RSTDST	15	*msbih.h*	reset destination BIC id
C_RUNDIAG	8	*msbih.h*	start specified diagnostics running
C_SELECT_ACT	13	*msbih.h*	select which MSBI is the current active unit
C_SETCONTROL	9	*msbih.h*	write BIC's control register
C_SETMODE	10	*msbih.h*	change current operational mode
C_START_EXEC	12	*msbih.h*	transfer control to specified address

(continued)

Table 8–1 AT&T Defined I/O Control Commands *continued*

Command	Value	Header File	Description
DEV_SUBD	0x193	*vdi_ioctl.h*	return all subdevices for a controller
DEV_TC	0x192	*vdi_ioctl.h*	return all the target controllers for a driver
DIAGOFF	('D' << 8 | 2)	*dsmd.h*	Turn all diagnostic reporting off */
DIAGON	('D' << 8 | 1)	*dsmd.h*	Turn diagnostic reporting on */
DIOC		*ioctl.h*	
DIOCGETB	'd' <<8 | 2	*ioctl.h*	
DIOCGETC	'd' <<8 | 1	*ioctl.h*	
DIOCSETE	'd' <<8 | 3	*ioctl.h*	
DMIOC		*dfdrv.h*	
D_BLANK	(('M' << 8) | 234)	*m050.h*	blank display
D_UNBLANK	(('M' << 8) | 235)	*m050.h*	unblank display
EDT_HEAD	0x191	*vdi_ioctl.h*	return the header of the EDT
EMPCHAN		*empath.h*	building block for empath constants
FIOC		*termfc.h*	
FORMAT	'f'	*diskette.h*	/* ioctl flag for format */
GETADDR	1	*ioadrv.h*	
GETEDT	7	*ioadrv.h*	
GETSTAT	8	*ioadrv.h*	
GETTYPE	6	*ioadrv.h*	
HA_VER	0X0083	*sdi.h*	get the host adapter version
HDECEREP	15	*hdeioctl.h*	clear error reports from the queue
HDECLOSE	9	*hdeioctl.h*	close hard disk
HDEERSLP	16	*hdeioctl.h*	wait (sleep) for an error report
HDEFIXLK	11	*hdeioctl.h*	"hdefix" locks hde log access
HDEFIXUL	12	*hdeioctl.h*	"hdefix" unlocks hde log access
HDEGEDCT	1	*hdeioctl.h*	get equipped disk count
HDEGEQDT	2	*hdeioctl.h*	get equipped disk table
HDEGERCT	13	*hdeioctl.h*	get count of outstanding error reports
HDEGEREP	14	*hdeioctl.h*	get outstanding error reports
HDEGETSS	4	*hdeioctl.h*	get sector size of disk
HDEMLOGR	10	*hdeioctl.h*	issue manual hdelog( ) requests
HDEOPEN	3	*hdeioctl.h*	open hard disk
HDERDISK	7	*hdeioctl.h*	read disk
HDERDPD	5	*hdeioctl.h*	read physical description of disk
HDEWDISK	8	*hdeioctl.h*	write disk
HDEWRTPD	6	*hdeioctl.h*	write physical description of disk
HXTIOCLINK		*vpmxt.h*	link channel 0
IBBNA	'I' <<8 | 17	*ib.h*	
IBCAC	'I' <<8 | 22	*ib.h*	
IBCLR	'I' <<8 | 27	*ib.h*	
IBCMD	'I' <<8 | 12	*ib.h*	
IBDINFO	'I' <<8 | 4	*ib.h*	
IBDMA	'I' <<8 | 30	*ib.h*	
IBEOS	'I' <<8 | 30	*ib.h*	
IBEOT	'I' <<8 | 30	*ib.h*	
IBGET	'I' <<8 | 0	*ib.h*	
IBGTS	'I' <<8 | 21	*ib.h*	
IBINB	'I' <<8 | 2	*ib.h*	
IBIOAB	'I' <<8 | 30	*ib.h*	
IBIST	'I' <<8 | 30	*ib.h*	
IBLLO	'I' <<8 | 40	*ib.h*	
IBLOC	'I' <<8 | 28	*ib.h*	
IBONL	'I' <<8 | 19	*ib.h*	

(continued)

Table 8–1 AT&T Defined I/O Control Commands *continued*

Command	Value	Header File	Description
IBOUTB	'I' <<8 | 3	*ib.h*	
IBPAD	'I' <<8 | 30	*ib.h*	
IBPCT	'I' <<8 | 29	*ib.h*	
IBPPC	'I' <<8 | 30	*ib.h*	
IBRD	'I' <<8 | 10	*ib.h*	
IBRDF	'I' <<8 | 40	*ib.h*	
IBRPP	'I' <<8 | 16	*ib.h*	
IBRSC	'I' <<8 | 30	*ib.h*	
IBRSP	'I' <<8 | 15	*ib.h*	
IBRSV	'I' <<8 | 20	*ib.h*	
IBSAD	'I' <<8 | 30	*ib.h*	
IBSET	'I' <<8 | 1	*ib.h*	
IBSGNL	'I' <<8 | 24	*ib.h*	
IBSIC	'I' <<8 | 30	*ib.h*	
IBSPOKE	'I' <<8 | 23	*ib.h*	
IBSRE	'I' <<8 | 25	*ib.h*	
IBTMO	'I' <<8 | 40	*ib.h*	
IBTRG	'I' <<8 | 26	*ib.h*	
IBWAIT	'I' <<8 | 18	*ib.h*	
IBWRT	'I' <<8 | 11	*ib.h*	
IBWRTF	'I' <<8 | 40	*ib.h*	
IBXTRC	'I' <<8 | 14	*ib.h*	
IBxxxx	'I' <<8 | 13	*ib.h*	
IFBCHECK	'F' <<8 | 2	*if.h*	Check memory address (64K boundary)
IFBCHECK	('F' << 8 | 2)	*if.h*	
IFCONFIRM	'F' <<8 | 3	*if.h*	Verify part of the format
IFCONFIRM	('F' << 8 | 3)	*if.h*	
IFFORMAT	'F' <<8 | 1	*if.h*	Format floppy disk
IFFORMAT	('F' << 8 | 1)	*if.h*	
IIOC		*ib.h*	
IOAINFO	2	*ioadrv.h*	
IOCTL_CNTRL(x)	(x >>3)&0x7	*had_ioctl.h*	Controller from PT minor number
IOCTL_DPRINTOFF	0x0110	*mz74.h*	turn on selected information prints
IOCTL_DPRINTON	0x0111	*mz74.h*	turn off selected information prints
IOCTL_DTRACEOFF	0x010	*mz74.h*	turn off function entry, exit, and progress points
IOCTL_DTRACEON	0x010	*mz74.h*	turn on function entry, exit, and progress points
IOCTL_GMINOR	0xff	*had_ioctl.h*	General use minor number for PT
IOCTL_HA(x)	(x >>6)&0x1	*had_ioctl.h*	HA from pass through minor number
IOCTL_HC(x)	(x >>3)&0xf	*had_ioctl.h*	HA/controller from PT minor number
IOCTL_LU(x)	x&0x7	*had_ioctl.h*	LU from PT minor number
I_CLRBIGB	41	*lo.h*	

(continued)

Table 8–1 AT&T Defined I/O Control Commands *continued*

Command	Value	Header File	Description
I_CLRWOFF	53	*lo.h*	
I_DDARG	22	*lo.h*	
I_ERRNAK	23	*lo.h*	
I_ERROR	25	*lo.h*	
I_FDINSERT	'S' <<8 | 020	*stropts.h*	
I_FIND	'S' <<8 | 013	*stropts.h*	
I_FLUSH	'S' <<8 | 05	*stropts.h*	
I_FREE	51	*lo.h*	
I_GETSIG	'S' <<8 | 012	*stropts.h*	
I_GRAB	50	*lo.h*	
I_GRDOPT	'S' <<8 | 07	*stropts.h*	
I_INTART	21	*lo.h*	
I_LINK	'S' <<8 | 014	*stropts.h*	
I_LOOK	'S' <<8 | 04	*stropts.h*	
i_MODCMD	30	*lo.h*	
I_NOARG	20	*lo.h*	
I_NREAD	'S' <<8 | 01	*stropts.h*	
I_PEEK	'S' <<8 | 017	*stropts.h*	
I_POP	'S' <<8 | 03	*stropts.h*	
I_PUSH	'S' <<8 | 02	*stropts.h*	
I_REVFD	'S' <<8 | 022	*stropts.h*	
I_SENDFD	'S' <<8 | 021	*stropts.h*	
I_SETBIGB	40	*lo.h*	
I_SETERR	43	*lo.h*	
I_SETHANG	42	*lo.h*	
I_SETOFAIL	44	*lo.h*	
I_SETSIG	'S' <<8 | 011	*stropts.h*	
I_SETWOFF	52	*lo.h*	
I_SLOW	28	*lo.h*	
I_SRDOPT	'S' <<8 | 06	*stropts.h*	
I_STR	'S' <<8 | 010	*stropts.h*	
I_TIMOUT	24	*lo.h*	
I_TRCLOG	1	*strlog.h*	process is tracer
I_UDARG	26	*lo.h*	
I_UDARGB	27	*lo.h*	
I_UNLINK	'S' <<8 | 015	*stropts.h*	
JAGENT	'j' <<8 | 9	*jioctl.h*	Control for both directions
JBOOT	'j' <<8 | 1	*jioctl.h*	
JMPX	'j' <<8 | 3	*jioctl.h*	
JTERM	'j' <<8 | 2	*jioctl.h*	
JTIMO	'j' <<8 | 4	*jioctl.h*	Timeouts in seconds
JTIMOM	'j' <<8 | 6	*jioctl.h*	Timeouts in milliseconds
JTRUN	'j' <<8 | 10	*jioctl.h*	Send runlayer command to layers

continued

Table 8–1 AT&T Defined I/O Control Commands *continued*

Command	Value	Header File	Description
LIOCSETS	'I' <<8 | 6	*ioctl.h*	
JTYPE		*jioctl.h*	
JWINSIZE	'j' <<8 | 5	*jioctl.h*	
JZOMBOOT	'j' <<8 | 7	*jioctl.h*	
LDCHG	'D' <<8 | 2	*termio.h*	
LDCLOSE	'D' <<8 | 1	*termio.h*	
LDGETT	'D' <<8 | 8	*termio.h*	
LDIOC		*termio.h*	
LDOPEN	'D' <<8 | 0	*termio.h*	
LDSETT	'D' <<8 | 9	*termio.h*	
LIOC		*ioctl.h*	
LIOCGETP	'I' <<8 | 1	*ioctl.h*	
LIOCGETS	'I' <<8 | 5	*ioctl.h*	
LIOCSETP	'I' <<8 | 2	*ioctl.h*	
LOAD	1	*sadldrv.h*	
LOADOSRTN	9	*ioadrv.h*	
LOCKED	000000002	*vpmtty.h*	lock for multiprocess running on a port
L_XRAM	0x142	*vdi_ioctl.h*	load XASRAM with the pattern
MIRR		*mirror.h*	
NIERRNO	'3' <<8 | 5	*ni.h*	Error number
NIGETA	'3' <<8 | 2	*ni.h*	Get value from Ethernet® header
NISETA	'3' <<8 | 1	*ni.h*	Set value from Ethernet header
PPC_VERS	'v' <<8 | 1	*ppc.h*	request version number of a ppc board (ioctl) */
PUMP	'p' <<8 | 8	*pump.h*	
PU_DLD	1	*pump.h*	
PU_EQUIP	6	*pump.h*	(not used)
PU_FCF	3	*pump.h*	
PU_GAD	4	*pump.h*	(not used)
PU_RST	2	*pump.h*	
PU_SYSGEN	5	*pump.h*	
RDBUF	'3' <<8 | 4	*ni.h*	Shared memory supply buffer
RTNADDR	5	*ioadrv.h*	
R_VME	0x111	*vdi_ioctl.h*	subcommand to read a target device on VMEbus
SDI_BRESET	0X0084	*sdi.h*	reset the SCSI bus
SDI_RELEASE	0X0086	*sdi.h*	release the device
SDI_RESERVE	0X0085	*sdi.h*	reserve the device
SDI_RESTAT	0X0087	*sdi.h*	device reservation status
SDI_SEND	0X0081	*sdi.h*	send a SCSI command
SDI_TRESET	0X0082	*sdi.h*	reset a target controller
SD_CHAR		*sd01_ioctl.h*	
SHA_REINIT	0xff	*had_ioctl.h*	Reinitialize the drive
SHA_RSTATE	0xfd	*had_ioctl.h*	Read a device state (3B4000 only)
SHA_WSTATE	0xfe	*had_ioctl.h*	Write a device state (3B4000 only)
SM_DISMM	0x161	*vdi_ioctl.h*	take vdi driver out of diagnostic mode
SM_ENAMM	0x162	*vdi_ioctl.h*	put vdi driver in diagnostic mode
SM_SRTSYS	0x165	*vdi_ioctl.h*	indicate that all VME subsystems should be started
SM_SRTVBUS	0x163	*vdi_ioctl.h*	indicate that this VME subsystem should be restored
SM_STPVBUS	0x164	*vdi_ioctl.h*	subcommand to stop the VME bus subsystem
STGET	"X" <<8 | 0	*stermio.h*	get line options
STR		*stropts.h*	
STSET	"X" <<8 | 1	*stermio.h*	set line options
STTHROW	"X" <<8 | 2	*stermio.h*	throw away queued input
STTSV	"X" <<8 | 4	*stermio.h*	get all line information
STWLINE	"X" <<8 | 3	*stermio.h*	get synchronous line number
SUPBUF	'3' <<8 | 3	*ni.h*	Shared memory supply buffer

(continued)

Table 8–1 AT&T Defined I/O Control Commands *continued*

Command	Value	Header File	Description
SXTIOCBLK	'b' <<8 | 5	*sxt.h*	
SXTIOCLINK		*sxt.h*	
SXTIOCLINK		*vpmsxt.h*	c type
SXTIOCNOTRACE	'b' <<8 | 2	*sxt.h*	
SXTIOCSTAT	'b' <<8 | 7	*sxt.h*	
SXTIOCSWTCH	'b' <<8 | 3	*sxt.h*	
SXTIOCTRACE	'b' <<8 | 1	*sxt.h*	
SXTIOCUBLK	'b' <<8 | 6	*sxt.h*	
SXTIOCWF	'b' <<8 | 4	*sxt.h*	
TCDSET	'T' <<8 | 32	*termio.h*	
TCFLSH	'T' <<8 | 7	*termio.h*	
TCGETA	'T' <<8 | 1	*termio.h*	
TCSBRK	'T' <<8 | 5	*termio.h*	
TCSETA	'T' <<8 | 2	*termio.h*	
TCSETAF	'T' <<8 | 4	*termio.h*	
TCSETAW	'T' <<8 | 3	*termio.h*	
TCSONC	'T' <<8 | 6	*termio.h*	
TCSSICTL	'T' <<8 | 64	*vpmtty.h*	pass 1 if set ctl, 0 is norm
TCTTIPMP	'T' <<8 | 65	*vpmtty.h*	pump BCT500; also pass pump
TIMOD		*timod.h*	
TIOC		*termio.h*	
tIOC		*ttold.h*	
TIOCEXCL		*vpmsxt.h*	exclusive cmd
TIOCEXCL		*vpmxt.h*	exclusive cmd
TIOCGETP	't' <<8 | 8	*ttold.h*	
TIOCNXCL		*vpmsxt.h*	non-exclusive cmd
TIOCNXCL		*vpmxt.h*	non-exclusive cmd
TIOCSETP	't' <<8 | 8	*ttold.h*	
TI_BIND	'T' <<8 | 102	*timod.h*	
TI_GETINFO	'T' <<8 | 100	*timod.h*	
TI_OPTMGMT	'T' <<8 | 101	*timod.h*	
TI_UNBIND	'T' <<8 | 103	*timod.h*	
TRCIOC		*trace.h*	
TTYTYPE	'T' <<8 | 8	*termio.h*	(3b15 only)
T_EOD	17	*st01_ioctl.h*	space to end-of-data
T_ERASE	15	*st00_ioctl.h*	erase medium
T_ERASE	15	*st01_ioctl.h*	erase medium
T_ERRLOG	2	*strlog.h*	process is error logger
T_LOAD	10	*st00_ioctl.h*	load medium
T_LOAD	10	*st01_ioctl.h*	load medium
T_LOCK	12	*st00_ioctl.h*	physically lock medium in driver
T_LOCK	12	*st01_ioctl.h*	physically lock medium in driver
T_RETENSION	16	*st01_ioctl.h*	tape retension
T_REVDIR	6	*st00_ioctl.h*	read reverse (not supported)
T_REVDIR	6	*st01_ioctl.h*	read reverse (not supported)
T_REVDIR	6	*tape_ioctl.h*	read reverse (not supported)
T_RWD	5	*st00_ioctl.h*	rewind to beginning of tape
T_RWD	5	*st01_ioctl.h*	rewind to beginning of tape
T_RWD	5	*tape_ioctl.h*	rewind to beginning of tape
T_SBB	4	*st00_ioctl.h*	space blocks backwards
T_SBB	4	*st01_ioctl.h*	space blocks backwards
T_SBB	4	*tape_ioctl.h*	space blocks backwards
T_SBF	3	*st00_ioctl.h*	space blocks forward
T_SBF	3	*st01_ioctl.h*	space blocks forward

(continued)

Table 8–1 AT&T Defined I/O Control Commands *continued*

Command	Value	Header File	Description
T_SBF	3	*tape_ioctl.h*	space blocks forward
T_SFB	2	*st00_ioctl.h*	space filemarks backwards
T_SFB	2	*st01_ioctl.h*	space filemarks backwards
T_SFB	2	*tape_ioctl.h*	space filemarks backwards
T_SFF	1	*st00_ioctl.h*	space filemarks forward
T_SFF	1	*st01_ioctl.h*	space filemarks forward
T_SFF	1	*tape_ioctl.h*	space filemarks forward
T_SFMB	8	*st00_ioctl.h*	space sequential filemarks backwards
T_SFMB	8	*st01_ioctl.h*	space sequential filemarks backwards
T_SFMF	7	*st00_ioctl.h*	space sequential filemarks forward
T_SFMF	7	*st01_ioctl.h*	space sequential filemarks forward
T_TRKSEL	14	*st00_ioctl.h*	move head to selected cartridge tape
T_TRKSEL	14	*st01_ioctl.h*	move head to selected cartridge tape
T_UNLOAD	11	*st00_ioctl.h*	unload medium
T_UNLOAD	11	*st01_ioctl.h*	unload medium
T_UNLOCK	13	*st00_ioctl.h*	physically unlock medium in driver
T_UNLOCK	13	*st01_ioctl.h*	physically unlock medium in driver
T_WFM	9	*st00_ioctl.h*	write filemarks
T_WFM	9	*st01_ioctl.h*	write filemarks
VERIFY	'v'	*diskette.h*	/* mode is 'v' to verify, 0 otherwise */
VIOC		*vtoc.h*	
VPMT		*vpmt.h*	
V_AL_TXLREG	0x170	*vdi_ioctl.h*	allocated dma segment translation registers
V_BREDT	B_REDT	*vdi_ioctl.h*	return edt for **getedt** command
V_CLRINT	0x200	*vdi_ioctl.h*	clear the interrupts
V_FORMAT	'V'<<8 | 6	*vtoc.h*	Get formatting parameters
V_GETFORMAT	'V'<<8 | 7	*vtoc.h*	Get PD values
V_GETINT	0x210	*vdi_ioctl.h*	return interrupt registers
V_GETMODE	0x1e0	*vdi_ioctl.h*	get vdi driver mode
V_GETSSZ	'V'<<8 | 5	*vtoc.h*	Get sector size for current disk.
V_HA	0x101	*vdi_ioctl.h*	subcommand to read/write the IOE
V_INIT_SC	0x1f0	*vdi_ioctl.h*	initialize the SC
V_INIT_XRAM	0x140	*vdi_ioctl.h*	initialize the SC XASRAM
V_PDREAD	'V'<<8 | 3	*vtoc.h*	Read Physical Description area
V_PDSETUP	'V'<<8 | 8	*vtoc.h*	Set PD values without writing to disk
V_PDWRITE	'V'<<8 | 4	*vtoc.h*	Write Physical Description area
V_POSTINTR	0x180	*vdi_ioctl.h*	post an interrupt to a VME device
V_PREAD	'V'<<8 | 1	*vtoc.h*	Physical read.
V_PWRITE	'V'<<8 | 2	*vtoc.h*	Physical write
V_RD_WRT	0x150	*vdi_ioctl.h*	issue read and write to host adaptor
V_READ_ADP	0x100	*vdi_ioctl.h*	read host adapter
V_RETEDT	0x190	*vdi_ioctl.h*	return EDT table information
V_SC	0x102	*vdi_ioctl.h*	subcommand to read/write the System Controller
V_SETMODE	0x160	*vdi_ioctl.h*	set VMEbus stae
V_TRAN_VME	0x110	*vdi_ioctl.h*	read from the VMEbus
V_VTOP	0x220	*vdi_ioctl.h*	return physical address for a supplied virtual address
V_WRT_ADP	0x120	*vdi_ioctl.h*	write to host adapter
W_VME	0x131	*vdi_ioctl.h*	subcommand to write to a target device on VMEbus
XERO_RAM	0x141	*vdi_ioctl.h*	zero the XASRAM
XGETADDR	3	*ioadrv.h*	

(continued)

Table 8–1 AT&T Defined I/O Control Commands *continued*

Command	Value	Header File	Description
XLOADSC	4	*ioadrv.h*	
XTIOCDATA	'b' <<8 | 5	*xt.h*	
XTIOCLINK	'b' <<8 | 1	*xt.h*	
XTIOCLINK	'b' <<8 | 6	*xt.h*	
XTIOCLINK		*vpmxt.h*	link channel 0
XTIOCNOTRACE	'b' <<8 | 4	*xt.h*	
XTIOCSTATS	'b' <<8 | 2	*xt.h*	
XTIOCTRACE	'b' <<8 | 3	*xt.h*	
XTIOCTYPE		*vpmxt.h*	c type
XTIOCTYPE		*xt.h*	

Using I/O Control Commands With Remote File Sharing

UNIX System V Release 3 includes the Remote File Sharing (RFS) utility that allows a process on one machine to access a file on another machine as if it were local. A *heterogeneous environment* is one in which RFS or a similar facility links machines with different architecture. I/O control commands that are accessed by a machine that uses different byte ordering and word size will not work and may corrupt the system. Note that the architectures of the SBC, 3B2, 3B15, and 3B4000 computers are similar, so accessing devices that use I/O control commands over an RFS network of these devices should not cause problems. However, if you are using RFS network to connect machines running different releases of UNIX System V, you may need to link the software against the system headers on the server machine to get the expected results.

When working with non-System V implementations of the UNIX system, advertising devices that use I/O control commands in an RFS network may not be advisable.

Chapter 9: Synchronizing Hardware and Software Events

Contents

Introduction

This chapter describes the use of functions provided by the UNIX operating system to synchronize hardware and software events. It provides information on the following:

- using the **sleep**(D3X) and **wakeup**(D3X) function pair
- using the **iowait**(D3X) and **iodone**(D3X) functions in block drivers
- using the **timeout**(D3X) and **untimeout**(D3X) functions
- using the **delay**(D3X) function
- using system time constants

Event Synchronization and Driver Development

Synchronizing hardware and software events concerns five areas of driver development.

- using **sleep**(D3X)/**wakeup**(D3X) to wait for an event
- using **iowait**(D3X)/**iodone**(D3X) to wait for an event
- using **timeout**(D3X)/**untimeout**(D3X) to delay the execution of a function
- using **delay**(D3X) to put a user process to sleep for a specified time
- using the built-in time constants

Table 9-1 summarizes how these functions are used:

Function(D3X)	Description	Level
delay(*ticks*)	Delay execution for *ticks* clock ticks	Base Only
iodone(*bp*)	Signal I/O completion	Base or Interrupt
iowait(*bp*)	Suspend execution during block I/O	Base Only
sleep(*event, priority*)	Suspend execution until *event*	Base Only
timeout(*function, arg, ticks*)	Call function in *ticks* clock ticks	Base or Interrupt
untimeout(*id*)	Cancel **timeout** with matching *id*	Base or Interrupt
wakeup(*event*)	Resume suspended execution	Base or Interrupt

Table 9–1 Synchronization Function Summary

The Level column indicates from which execution level the function can be called.

CAUTION: The **sleep**, **iowait**, and **delay** functions must never be called from an **init** or interrupt routine. Called from an **init** routine, the computer hangs when booted. Called from an interrupt routine, an unknown process is put to sleep with no mechanism for wakeup.

Waiting for an Event

An important component of the driver data movement concerns how drivers wait for and respond to certain hardware or software events. Usually, waiting for an event is a result of different hardware and software execution speeds. The waiting functions are called under three circumstances.

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

Waiting For Hardware

By human terms, the time required for a device such as a disk drive or terminal to perform some action seems instantaneous. Actually the CPU is operating much faster than the device and the time required by the device seems interminable. A waiting function is required to release the CPU from wasting precious fractions of seconds waiting for a device to complete an action. The functions used to wait for a hardware action are the **iowait** and **sleep**. **iowait** is only used to suspend processing in a block driver when waiting for buffered I/O to complete. **sleep** is used for any type of driver.

The computer is designed so that when a device has a block of data ready to be transferred, the device sends a cue (called an interrupt) to the operating system to tell it to call a driver interrupt routine to fetch the data. The operating system keeps track of which driver is associated with the device generating the interrupt and calls the proper driver interrupt routine. While the interrupt routine call is automatic, a command required to resume execution of a suspended process must be handled by the driver. When execution is suspended with **iowait**, **iodone** must be called to restart process execution; when **sleep** is called to suspend execution, **wakeup** is called to resume execution.

Technically, **sleep** could be called instead of **iowait**, but **iowait** is a convenience for working with the system buffer cache for these reasons

- **iowait** executes a **while**-loop to check `bp->b_flags&B_DONE`
- **iowait** decrements **syswait.iowait**
- If `bp->b_flags&B_ERROR` is true, then **u.u_error** is set to `bp->b_error`, if a value is there, or set to EIO if not.

A negative with using **iowait** is that it executes **spl0** thereby enabling all interrupts.

iowait and **iodone** have as an argument a pointer to the `buf` structure (*bp*). **sleep** and **wakeup** use as their argument, an arbitrary address to guarantee that the **wakeup** call restarts the proper suspended process. (**sleep** has an additional argument which is explained later in this chapter.) Each of the event synchronization functions are described in separate sections in this chapter.

Waiting For Software

Use **geteblk** when requesting a buffer for a block driver or **getcb** for a character driver. Should a buffer not be readily available both functions sleep until one is available. When using a private buffering scheme and a buffer is not available, sleep on the last element of that structure.

Some functions provide an automatic **wakeup** function call. For example, **getc** and **putcf** both wake up processes that have called **sleep** to wait for a buffer on the character block free list, `cfreelist`. As a rule, though, unless so indicated in the function you are calling in the *BCI Driver Design Reference Manual*, a **wakeup** must be provided for every **sleep** call.

Waiting By Timing an Event

The "stopwatch" mode for timing an event requires specifying the number of time units that a process is to be suspended. This is useful for transferring data character-by-character such as when the hardware imposes a baud rate on your driver, or for retrying some event at a later time when a sleep on a device may not succeed. The **delay** and **timeout** functions are used to suspend a process for a specified length of time. **delay** suspends execution of the immediate process. **timeout** is used to execute a function after the time elapses. The difference between the two is that **timeout** returns immediately after scheduling the future event, and **delay** stops execution until the time elapses. The **untimeout** function is provided to stop a previously set **timeout**. (**timeout** returns an **int** identification number that is passed as the argument to **untimeout** to stop the previous call.) The time arguments for **delay** and **timeout** are generally expressed using the HZ constant which is equal to one second. For example, HZ/100 is one one-hundredth of a second, or HZ*2 is two seconds.

Using the Sleep and Wakeup Functions

The most common mechanism for waiting for an event to occur is the **sleep/wakeup** function pair. The driver issues an I/O request and then waits for it by calling the **sleep** function. While the driver is waiting, the system performs a context switch and starts another process executing. When the event (a system state in hardware or software) happens, an interrupt is generated that calls the interrupt routine in the driver. The **wakeup** function is called from the driver interrupt routine to resume the execution of the suspended process.

For example, when a **read**(2) request is made to obtain data from a disk drive, the disk drive does not have the capacity to deliver data as quickly as the request is made. Therefore, **sleep** must be called to suspend execution of the process while the data is fetched from the disk drive.

A sleeping process is still considered to be an active process, but is kept on a queue of jobs whose execution is suspended while they wait for a particular event. When the process goes to sleep it specifies the event that must occur before it may continue its task. The **sleep** call records the process number and the event, then places it on the list of sleeping processes. Control of the machine is then transferred to the highest-priority runnable process.

The **sleep** function requires two arguments: the address upon which the process will sleep, and a priority value that is assigned to the process when it is awakened:

```
sleep(addr, pri)
```

Interrupt handler routines should never call **sleep** since **sleep** affects the currently executing process, and a process independent of the device could be executing when the device interrupted. If the interrupt routine were to call **sleep**, the process that was interrupted would be put to sleep for reasons beyond its control. More importantly, in some UNIX system implementations, sleeping in an interrupt routine could cause the system to crash because of the interdependency of the process context switch mechanism and interrupt levels. The interrupt routine must therefore not invoke other functions that could lead to a call to **sleep**, such as **iowait** or **copyin/copyout**. See the reference pages for the interrupt routines in section D2X for a complete list of functions that cannot be called from an interrupt routine.

NOTE: Any **sleep** call with a corresponding **wakeup** in the interrupt routine, should be protected from interrupts with the **splhi** function to ensure that no interrupts occur when that section of code is being executed. Otherwise, the **wakeup** call could come before the process goes to **sleep**, in which case the process will never awaken. This is discussed later in this chapter.

Sleep Addresses

The first argument to the **sleep** function is an address that has no meaning except to the corresponding **wakeup** function call; addresses are used because their uniqueness is easy to control. The event should be an external (rather than a local) variable. If a process sleeps on a local variable, a chance is taken that the wrong process will awake or that the process associated with your driver will be awaken for the wrong reason.

The **sleep** addresses are usually taken from the entry in the device data structure of the device the process is accessing to guarantee uniqueness across the system. When a process **sleeps** on the device data structure, the driver should set a flag in that structure indicating the reason to sleep.

```
spl6()
        driver.state |= condition;
        sleep(&driver.state, PRIORITY);
splx()
```

A driver can sleep on other structures, such as `bfreelist` or `cfreelist`. When sleeping on `bfreelist`, set **B_WANTED** in the **b_flags** member of the buffer header. When sleeping on `cfreelist`, set **cfreelist.c_flag** to a positive value. When sleeping on a private buffering pool, you should sleep on the last element of that structure.

Waking Up a Sleeping Process

Either an interrupt handler or another process later calls the **wakeup** function to awaken the sleeping process. The **wakeup** function takes one argument: the address upon which the process was sleeping as set by the corresponding **sleep** function:

```
wakeup(addr)
```

The code invoking the **wakeup** function should check for a particular flag bit, indicating the reason that the process is sleeping. The driver then calls **wakeup** with one argument, namely the address where a process could be sleeping.

```
if (driver.state&condition)
        wakeup(&driver.state);
else
        ERROR;
```

There should be a one-to-one correspondence between events and **sleep** addresses; one address should not be used for sleeping for two events. This helps ensure kernel sanity, enhances driver efficiency and code readability. If several processes are sleeping for the same resource and do not have one-to-one correspondence, they may all be awakened at the same time, and the first to run will grab the resource. NOTE: This is desirable in some circumstances such as when two processes are reading the same disk block.

The **wakeup** function awakens all processes sleeping on the address, enabling them to execute when the scheduler chooses them. If no process is sleeping on the address when **wakeup** is called, **wakeup** returns without an error.

When a process receives a **wakeup** call, the driver may need to check that certain conditions are true before actually resuming execution. Checking conditions is important when more than one process is sleeping on the same address. You can use **while** or another programming loop to check for a certain condition, as shown in Figure 9-1.

```
1   /*
2   An example of a while loop for getting a resource.
3   If the resource is not available, sleep is called.
4   */


5   struct cblock *
6   alloccblock()
7   {
8           register struct cblock *bp;
9           register int s;

10          s = splhi();
11          while ((bp = cfreelist.c_next) == NULL) {
12                  cfreelist.c_flag = 1;
13                  sleep(&cfreelist);
14          }
15          cfreelist.c_next = bp->c_next;
16          bp->c_next = NULL;
17          bp->c_first = 0;
18          bp->c_last = cfreelist.c_size;
19          splx(s);
20          return(bp);
21  }
```

Figure 9–1 sleep — while Loop for Condition Testing

Table 9-2 lists functions that wake up processes sleeping on buffer list addresses. This information is useful for knowing which functions will wake up a process without need for your driver to call **wakeup**.

Table 9–2 wakeup Calls in Functions

Function(D3X)	Code

brelse	`if (bp->b_flags&B_WANTED)`
	`wakeup((caddr_t)bp);`
	
	`if (bfreelist.b_flags&B_WANTED) {`
	`bfreelist.b_flags &= B_WANTED;`
	`wakeup((caddr_t)&bfreelist);`
	`}`

getc, putcf	`if (cfreelist.c_flag) {`
	`cfreelist.c_flag = 0;`
	`wakeup(&cfreelist);`
	`}`

mfree	`if (mapwant(mp)) {`
	`mapwant(mp) = 0;`
	`wakeup((caddr_t)mp);`
	`}`

physio	`/* if a buffer was allocated, then wakeup`
	`* processes sleeping on pfreelist */`
	
	`<If a buffer was allocated, then:>`
	`spl6();`
	`bp->av_forw = pfreelist.av_forw;`
	`pfreelist.av_forw = bp;`
	`pfreecnt++;`
	`wakeup(&pfreelist);`
	`spl0();`
	`}`

Preventing Signals

The second argument to the **sleep** function is a scheduling parameter that controls when the process will be awakened from its sleep; this argument is usually a constant rather than a variable. The argument, called the sleep priority, has critical effects on the sleeping process's reaction to signals.

Priority values range between 0 (highest priority) and 39 (lowest system priority). You should use a defined constant for sleep priorities, either one of the standard ones or one you define yourself. Some priority constants are included in UNIX System V. Table 9-3 lists these.

Table 9–3 sleep Priority Levels

Constant	Value	Defined In	Used For
PRIBIO	20	*param.h*	Sleep priority for block devices
PZERO	25	*param.h*	Priority for deciding whether signals can awaken the process
TTIPRI	28	*tty.h*	Sleep priority for TTY device's input
TTOPRI	29	*tty.h*	Sleep priority for TTY device's output

Constants for your own driver should be defined either in the header file for your driver or in the global data structure section of the driver code itself. The declaration can assign either an absolute value or a value relative to PZERO. For instance,

```
#define DRVPRI 29
#define DRVPRI (PZERO + 4)
```

result in the same priority for the DRVPRI priority.

Block Driver iowait/iodone Event Synchronization

Block-access drivers using the buffer header buffer scheme that are waiting for an I/O event use the **iowait/iodone** pair instead of **sleep** and **wakeup**.

The **iowait** function can be used to put a block driver to sleep until the I/O operation is complete. **iowait** sleeps at a priority of 20 (PRIBIO). Since it operates on an I/O buffer header, it is not used by a character device (although it is used by a block devices doing raw I/O through **physio**).

iowait sets `b_flags` to B_READ, B_WRITE, or B_PHYS to indicate the type of operation and calls the **sleep** function. The interrupt routine should call the **iodone** function when the I/O is complete; **iodone** sets the `b_flag` member to B_DONE. If the **b_asynch** bit is set, the interrupt routine must call **brelse** to release the buffer.

timeout/untimeout Event Synchronization

In some cases a driver must be sure that it is awakened after a maximum period. For those situations where a limit must be placed on how long a process will sleep, the **timeout** facility is available.

The **timeout** function can be used in conjunction with **sleep** to ensure that the driver is awakened after a certain period of time. **timeout** can also be used alone to indicate that a driver function is to be called after a specified period of time. The **timeout** function can be canceled with the **untimeout** function.

NOTE: The function called by **timeout** is called from an interrupt mode. Therefore, functions that can't be executed from an interrupt routine cannot be called from **timeout**.

timeout is invoked as:

timeout(*function, function-argument, clock-ticks*)

The *function* argument can be any kernel function that can operate from an interrupt routine including **timeout** itself. *function-argument* is an argument to the *function*. If you do not need an argument for the function you are specifying, include any value, such as zero. Each argument must be specified. *clock-ticks* is the number of time units that the function will be delayed before executing. *clock-ticks* are usually specified as a multiple of HZ. HZ (defined in *param.h*) gives the clock frequency used by a given kernel.

A sample **timeout** call is

```
timeout(repeat, n, HZ);
```

where **n** is the argument to the function **repeat**, to be called after one second's worth of clock cycles. The exact time until the **timeout** takes effect may not be precise because of the interaction of other parts of the system. The compiler requires prior declaration of the function name argument to **timeout**, as in

```
extern char *repeat();
timeout(repeat, n, HZ);
```

depending where the function **repeat** is defined.

Using Timeout with Sleep

A driver can ensure that it will be able to resume its execution even if no call to **wakeup** is made by first calling **timeout** and then **sleep** This should be done, however, only if truly necessary, as it carries some heavy processing requirements. When the call to **timeout** is made, it inserts the specified event into the callout table. This data structure is a list of events in a simple array. Insertion of the event requires copying all elements of the list following the inserted event.

If the sleeping process is not awakened before the "timeout" event, the specified function is be called unless you have called **untimeout**. The second argument to the **timeout** routine could be the event the driver was about to sleep on. When the function is called, it can use this information to call **wakeup** to wake the driver. The function called from the callout table should also set some internal flag to permit the driver to distinguish between the two ways it can be awakened.

Using timeout For An Operator Request

Another use for the **timeout** function is in a driver that sends a message to the system console requesting that the operator take a certain action. For instance, the **write**(D2X) routine for a tape drive may have a section that tells the operator to mount a tape. Use the **sleep** function to suspend processing until the new tape is mounted. If a number of other console messages are generated, the message telling the operator to mount the tape could disappear from the screen before it is seen. By using a **while** statement in conjunction with **sleep**, the driver will continue to display the mount request on the console. Rather than have this message displayed continuously, the **timeout** function can specify how often to redisplay the message. Once the request is honored, the driver's interrupt routine cancels the **timeout** operation with the **untimeout** function.

The following routine called by an **open**(D2X) routine (starting in line 20 in Figure 9-2) illustrates this. After the input arguments have been verified, the status of the device is tested. If the device is not on-line, a message is displayed on the system console (line 39). The driver schedules a wakeup call with the **timeout** (line 41) and waits for 5 minutes (**sleep**). If the device is still not ready, the procedure is repeated.

When the device is made ready, an interrupt is generated (this assumes that the device was designed to generate an interrupt when a tape is mounted). The driver interrupt handling routine (line 53) notes there is a suspended process. It cancels the timeout request with **untimeout** (line 61) and wakens the suspended process (line 63).

```
1  struct  mtu_device              /* Layout of physical device registers */
2  {
3         int      control;       /* Physical device control word        */
4         int      status;        /* Physical device status word         */
5         int      byte_cnt;      /* Number of bytes to be transferred   */
6         paddr_t  baddr:         /* DMA starting physical address       */
7  }; /* end device */

8  struct mtu                     /* Magnetic tape unit logical structure */
9  {
10        struct buf *mtu_head;  /* Pointer to head of I/O queue         */
11        struct buf *mtu_tail;  /* Pointer to tail of buffer I/O queue  */
12        int        mtu_flag;   /* Logical status flag                  */
13        int        mtu_to_id;  /* Time out id number                   */
14        ...
15 }; /* end mtu */

16 extern struct mtu_device *mtu_addr[];/* Location of physical device registers */
17 extern struct mtu         mtu_tbl[]; /* Location of logical device structures */
18 extern int    mtu_cnt;
19  ...
20 mtu_openxx(dev, flag)
21 dev_t dev;
22 {
23 register struct mtu *dp;
24 register struct mtu_device *rp;

25     if  ((minor(dev) >> 3) > mtu_cnt) { /* If device does not exist,    */
26         u.u_error = ENXIO;              /* then return error condition  */
27         return;
28     } /* endif */
29     dp = &mtu_tbl[minor(dev)];          /* Get logical device struct    */
30     if  (dp->mtu_flag & MTU_BUSY) != 0) {  /* If device is in use,      */
31         u.u_error = EBUSY;              /* then return busy status      */
32         return;
33     } /* endif */
```

Figure 9–2 The timeout Function

```
34      dp->mtu_flag = MTU_BUSY; /* Indicate device is in use & clear other flags */
35      rp = xx_addr[minor(dev) >> 3];      /* Get device regs */
36      oldlevel2 = splhi();
37      while((rp->status & MTU_LOAD) == 0) /* While a tape is not loaded,      */
38      {                                   /* display mount request on console */
39          cmn_err(CE_NOTE, "!Tape MOUNT request for drive %d", minor(dev) & 0x3);
40          dp->mtu_flag |= MTU_WAIT;       /* Indicate process is suspended  */
41          dp->mtu_to_id = timeout(wakeup, dp, 5*60*HZ);   /* Wait 5 minutes */
42          if  (sleep(dp, (PCATCH | PZERO + 2)) == 1)  /* Wait for tape load   */
43          {                           /* If user aborts process, then         */
44              dp->mtu_flag = 0;  /* release tape device by clearing flags */
45              untimeout(dp->mtu_to_id);
46              splx(oldlevel2);
47              longjmp(u.u_qsav); /* Abort open(2) system call */
48           } /* endif */
49      } /* endwhile */
50      splx(oldlevel2);
51    ...
52  } /* end mtu_open */
53    ...
54  mtu_int(cntr)
55  int cntr;       /* Controller that caused the interrupt */
56  {
57  register struct mtu_device *rp = xx_addr[cntr];   /* Get device regs */
58  register struct mtu *dp = &mtu_tbl[cntr << 3 | (rp->status & 0x3)];
59    ...
60      if  ((dp->mtu_flag & MTU_WAIT) != 0)  /* If a process is suspended */
61      {                                     /* waiting for a tape mount, then*/
62          untimeout(dp->mtu_to_id);     /* cancel timeout request       */
63          dp->flag &= ~MTU_WAIT;        /* Clear wait flag              */
64          wakeup(dp);                   /* Awaken suspended process       */
65      } /* endif */
66    ...
```

Figure 9–3 The untimeout Function

Using the delay Function

This function is used to stop execution of the current process for a given period of time. Drivers can use the **delay** function instead of the **timeout** function, to instruct the driver to **sleep** for a specified amount of time and then **wakeup**.

To use **delay**, specify the amount of time to wait. **delay** automatically calls **wakeup** to resume execution.

Figure 9-4 illustrates the use of **delay**. This code is from a driver for a line printer. Before allocating buffers and storing data in them, the driver checks the status of the device (line 10). If the printer needs to have paper loaded, it displays a message on the system console (line 12). If the driver called **sleep** directly, the operator would have to signal when the paper was loaded. By using **delay**, the driver waits one minute (line 13) and tries again. If paper is loaded, processing will resume automatically.

```
1  struct  device                  /* Layout of physical device registers */
2  {
3          int      control;       /* Physical device control word        */
4          int      status;        /* Physical device status word         */
5          short    xmit_char;     /* Transmit character to device        */
6  }; /* end device */

7  extern struct device xx_addr[];   /* Location of physical device registers */
8   ...
9  register struct device *rp = &xx_addr[minor(dev) >> 4)]; /* Get device regs */

10 while(rp->status & NOPAPER)    /* While printer is out of paper */
11 {            /* display message & ring bell on system console */
12     cmn_err(CE_WARN, " xx_write: NO PAPER in printer %d 07", (dev & 0xf));
13     delay(60 * HZ);            /* Wait one minute and try again */
14 } /* endwhile */
```

Figure 9–4 delay — Allows Manual Intervention

Time Constants

The UNIX operating system provides a set of constants that are updated by the system clock interrupt. The clock ticks every 10 milliseconds on all computers referenced in this book except the 3B4000 ADP. The clock on the 3B4000 ADP ticks every 50 milliseconds. **lbolt** contains the number of seconds since the last system boot. **time** contains the number of seconds since 00:00:00 GMT (Greenwich Mean Time) January 1, 1970. HZ is provided to indicate the value of one second. The UNIX operating system clock is accurate to within plus or minus five clock ticks. Therefore, the time can never be determined exactly.

- HZ — (hertz)† is one second. HZ is defined in *param.h*.
- **lbolt** — (lightning bolt) is updated by the kernel each tick and represents the time in ticks since the last boot. **lbolt** is a **time_t** (**long**) data type. Note that as previously mentioned, **lbolt** is updated five times slower on the 3B4000 ADP than on any other AT&T computer referenced in this book.
- **time** — the time in seconds since 00:00:00 (GMT) January 1, 1970. **time** is a **time_t** (**long**) data type and is updated once every second.

† HZ is an abbreviation for hertz. However, HZ has no association with the electrical notation "hertz."

H Z

HZ is a defined constant found in *param.h* which specifies the number of clock ticks per seconds on a given machine. HZ is normally used in calling the **timeout** function for some amount of time, since the time passed to **timeout** is given in ticks and HZ is set to the number of ticks in a second.

For example, the **tttimeo** function uses HZ to determine how many ticks to delay when a driver has requested non-canonical processing with t_cc[VTIME] tenths of seconds waiting period. HZ/10 is the number of ticks in a tenth of a second.

Refer to Figure 9-5 for another usage example.

```
1  /* scan xx device for input every second */
2  xxscan()
3  {
4  /* scan for input */
5  /* call xxscan after 1 second */
6  timeout(xxscan,0,HZ);
7  }
```

Figure 9–5 HZ — Usage Example

lbolt

lbolt is a system external integer of the number of ticks since the last system boot. This value may be used as a counter for driver response time. **lbolt** is used to save a starting time for some driver operation, and then compared with the **lbolt** value once the operation is over to get a response time for the operation.

Figure 9-6 shows how **lbolt** is used to time an I/O operation.

```
1   #include "sys/types.h"

2   extern time_t lbolt;
3   struct xxstat xxstat; /* stats about xx device I/O */

4   xxstrategy(bp)
5   struct buf *bp;
6   {

7   /* schedule I/O for xx device */

8           xxstat.begintime = lbolt;
9   }

11   xxint(dev)
12   {

13          /* determine which interrupt came through and which operations
14          were completed */

15          xxstat.endtime = lbolt;
16          xxstat.operationtime = xxstat.endtime - xxstat.begintime;
17          xxstat.totaltime += xxstat.operationtime;
18          xxstat.operations++;

19          if (xxstat.operations > 0)
20              xxstat.avgtime = xxstat.totaltime / xxstat.operations;
21  }
```

Figure 9–6 lbolt — Timing an I/O Operation

time

time is an external integer set to the number of seconds since 1/1/70 00:00:00 GMT. It is updated once each second by the system clock. **time** may be used when any timing in seconds needs to be done, or when the time of the last update on a structure needs to be stored.

The following example shows the use of **time** for timing an I/O operation in a driver **write** routine.

```
1   extern time_t time;

2   struct datalog datalog;

3   xxwrite(dev)
4   {
5   /* update data to device  or structure */
6   datalog.start_time_in_secs = time;
7   /* do I/O */
8   datalog.time_of_last_IO = time - datalog.start_time_in_secs;
9   datalog.lastupdatetime = time;
10  }
```

Figure 9–7 time — Timing an I/O Operation

Chapter 10: Interrupt Routines

Contents

Introduction

This chapter introduces interrupt handling in the UNIX operating system, and provides guidelines on writing interrupt handling routines for both character and block devices. The following general topics are discussed:

- interrupt vectors, how the *interrupt vector table* is accessed, and how interrupt vector numbers are assigned to specific interrupt vectors
- how the operating system services interrupts
- writing **int**, **rint**, and **xint** interrupt routines for intelligent and non-intelligent character and block devices
- using the **spl*** set of functions to set processor priority levels and protect critical sections of driver code

Interrupts and the UNIX Operating System

An interrupt is any service request that causes the CPU to stop its current execution stream and to execute an instruction stream that services the interrupt. When the CPU finishes servicing the interrupt, it returns to the original stream and resumes execution at the point it left off. Interrupts are requested from one of the three following sources:

- hardware devices
- software interrupts (Programmed Interrupt Requests or PIRs)
- exceptions such as page faults

Hardware devices use interrupt requests to signal a range of conditions including: successful device connections, write acknowledgements, data availability, and read/write completions. The CPU is responsible for associating the interrupt request with a specific driver interrupt routine using entries in an internal table called the *interrupt vector table*[1]. The driver's interrupt routine determines the reason for the interrupt, services the interrupt, and wakes up any base level processes waiting on the interrupt completion. For example, when a disk drive is ready to transfer information to the host to satisfy a read request, the disk drive generates an interrupt. The CPU acknowledges the interrupt and calls the disk driver's interrupt routine. The driver interrupt routine then wakes up the process waiting for data which conveys the data to the user.[2]

AT&T computers that use a WE 32000 series microprocessor accept fifteen levels of interrupts. The level indicates the degree of priority given the interrupt by the CPU. The higher the priority, the quicker the system will service the interrupt when multiple interrupts are pending. Level zero is the highest priority, level 14 is the lowest. Level 15 indicates that no interrupts are pending. The Interrupt Priority Level (IPL) for the requesting device is determined by the device itself and is entered in the device driver's master file under the IPL column.[3]

The following sections discuss the types of interrupt requests the CPU processes.

Hardware Interrupts

For hardware devices, interrupts are the primary method of communication with the CPU. Hardware interrupts tell the CPU that a read or write have been completed, or that a character has been received or transmitted.

---

2. See "Interrupt Vectors" in this chapter for information on the interrupt vector table.
2. Refer to *Maxicomputing in Microspace*, (referenced in Chapter 1) for a detailed explanation of how interrupts are initiated and acknowledged.
3. See Chapter 3 in this book for a description of the master file.

The driver writer is responsible for writing the interrupt portion of the device's driver. UNIX provides a few generic interrupt handling routines for hardware interrupts, but the driver writer has to supply the specifics about the particular device. Some devices send only one type of interrupt and the interrupt routine must be responsible for determining the kind of interrupt sent. Other devices, primarily TTY devices, send two types of interrupts: one receive and one transmit.

In general, an **int(D2)** routine should be written for any device that does not send separate transmit and receive interrupts. TTY devices that do request separate transmit and receive interrupts have two separate routines associated with them: **xint**(D2), for a transmit interrupt, and **rint**(D2), for a receive interrupt.[4]

Not all hardware devices send interrupt requests directly to the CPU. Some device interrupts are first handled by an intermediary interrupt routine that is part of an intermediary driver. Devices that must first send their interrupts through an intermediary interrupt handler are called *external devices*. For example, on the 3B4000 computer, interrupts sent by SCSI devices supported by an extended SCSI bus are first captured by firmware on the SCSI bus host adapter called a SLIC. The host adapter then issues an interrupt request to the CPU. The CPU then associates the interrupt with one interrupt routine for the host adapter. The identity of the specific device that originally issued the interrupt request is passed through the *ivec* argument to the interrupt routine.[5]

Software Interrupts

In addition to the hardware interrupts discussed in this chapter, the AT&T computers support software interrupts called Programmed Interrupt Requests (PIRs). A PIR is generated by writing an integer into a logical register address assigned to the interrupt vector table.

PIRs are seldom used for drivers other than those developed as part of the operating system itself, and so are not discussed here. To establish a PIR, you must modify the system initialization software and run extensive tests on the bootstrap software to ensure that the PIR is not corrupting the system timing mechanism and interrupt vectors.

Exceptions

Exceptions are error conditions that interrupt the current processing of the CPU and require special fault handler processing for recovery. Fault handlers are responsible for executing instructions to handle the specific fault, and for restarting the interrupted instruction sequence once the fault is handled. Like device interrupts, exceptions are associated with their fault handlers through a separate exception vector table.

---

4. See "Writing Interrupt Routines" in this chapter for more information.
5. See "The Interrupt Routine Argument" in this chapter for information on the *ivec* argument.

The following three types of events cause exceptions:

- **Internal faults** - error conditions detected by the processor during an instruction sequence.
- **External faults** - error conditions detected outside the processor and conveyed to it over its fault input.
- **Traps** - internal error conditions detected by the processor at the end of an instruction.

It is not the responsibility of the driver writer to account for exceptions that may occur in the system. However, it is important to note that exceptions contend with device interrupt requests for the use of the CPU.

Interrupt Vectors

An interrupt vector is an entry to a table, called the *interrupt vector table*, that is assigned to an interrupt when the system is booted. The interrupt vector table resides in kernel space in main memory and associates interrupts with their appropriate interrupt routines. Every device that is not external has at least one interrupt vector table entry. Each entry is assigned an *interrupt vector number* that associates the interrupt with the text address identifying the starting address of the interrupt handler for that interrupt. When an interrupt occurs, the CPU associates the interrupt with its interrupt vector number, fetches the starting address of the interrupt handler, and executes the address to service the interrupt.

The #VEC column of a driver's master file determines the number of interrupt vectors required for the device the driver supports. When the system boots, the #VEC column is accessed, and the appropriate number of interrupt vector table entries are created for that device. The AT&T computers referenced in this book can support up to 256 interrupt vector table entries.

Not all devices need interrupt vectors for every interrupt they request. Most disk controllers for 3B computers that support multiple devices have the capability of interpreting the interrupts issued by each subdevice. Therefore, the controller for these devices only then sends one interrupt to the CPU. Other devices, such as serial ports that each generate transmit and receive interrupts, have separate interrupt vectors for transmit and receive.

Interrupt Vectors and System Initialization

The system initialization program, **lboot**, runs when the system is booted and reads the #VEC field in the driver's master file to determine the number of interrupt vectors per controller and assigns numbers accordingly. The CPU uses these vector number assignments to associate the interrupt with the appropriate interrupt handler routine. **lboot** compares the value in the #DEV (number of subdevices) column to the value in the #VEC (number of vectors) column to determine whether the driver requires an **int**(D2) routine or the **rint**(D2X)/**xint**(D2X) pair of routines. If the value of #VEC is double the value of #DEV (indicating that each subdevice has two interrupt vectors), **lboot** assumes **rint** and **xint** routines are being used; otherwise, **lboot** assumes an **int** routine is being used. **lboot** assigns what it deems to be the appropriate interrupt handler for the #VEC-to-#DEV ratio regardless of what is coded for the driver. If the proper routines (**rint/xint** or **int**) have not been coded, interrupts received for the device will be spurious and may corrupt another driver or crash the system.

Interrupt Vector Number Assignment

Entries for most devices in the interrupt vector table are assigned transparently by the system. Driver writers do not need to know how numbers are assigned by the system. However, some devices require their vector numbers hardcoded in the driver master file. The following section discusses these devices. This section is provided primarily for your interest.

For 3B2, 3B15, and 3B4000 systems, the system automatically generates vector numbers in groups of 16 for each device that is listed in the Equipped Device Table (EDT). The first vector assigned to a device (controller) is determined by multiplying the external major number (board code) by 16. Subsequent vectors count up from there. Note that this imposes a limit of 16 subdevices per controller unless the device has the intelligence necessary to associate interrupts with a subdevice in some way other than the interrupt vectors.[6]

If each controller has only one interrupt vector, its number is:

ext-major-number * 16

If each subdevice has one interrupt vector, each number is determined by the formula:

(*ext-major-number* * 16) + *subdevice-number*.

Consider the configuration in Figure 10-1 of one driver controlling two devices (controllers), each of which has four subdevices.

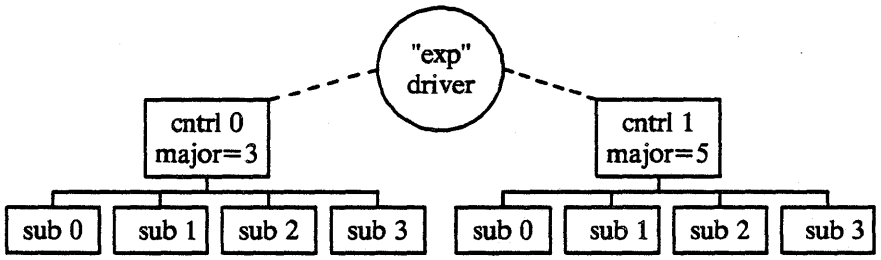


Figure 10–1 Sample Configuration

5. See "Interrupt Vector Number Assignment" for more information.
6. All devices discussed in this book require interrupt vectors.

Table 10-1 gives the interrupt vectors assigned for the sample configuration if each subdevice has one interrupt vector.[7]

Table 10–1 Subdevices With One Interrupt Vector

Master File Values: #VEC=4 #DEV=4				
				vector
controller	**subdev**	*ivec*	**number**	**equation**
0	0	0	48	(3 * 16) + 0
(major=3)	1	1	49	(3 * 16) + 1
	2	2	50	(3 * 16) + 2
	3	3	51	(3 * 16) + 3
1	0	4	80	(5 * 16) + 0
(major=5)	1	5	81	(5 * 16) + 1
	2	6	82	(5 * 16) + 2
	3	7	83	(5 * 16) + 3

If each subdevice supports two interrupt vectors (meaning the driver must use the **rint/xint** routines), the vectors are divided into transmit and receive portions. Table 10-2 gives the interrupt vectors assigned for the configuration if each subdevice has eight interrupt vectors.

7. The figures listed in this section include entries for the *ivec* argument. The *ivec* argument is passed to the interrupt routine as a means of identifying the specific device or subdevice requesting the interrupt. See the "The Interrupt Routine Argument" section in this chapter for more information.

Master File Values: #VEC=8 #DEV=4				
controller	**subdev**	*ivec*	**vector**	**vector portion**
0	0	0	48	0 (transmit)
(major=3)		1	49	1 (receive)
	1	2	50	0 (transmit)
		3	51	1 (receive)
	2	4	52	0 (transmit)
		5	53	1 (receive)
	3	6	54	0 (transmit)
		7	55	1 (receive)
1	0	8	80	0 (transmit)
(major=5)		9	81	1 (receive)
	1	10	82	0 (transmit)
		11	83	1 (receive)
	2	12	84	0 (transmit)
		13	85	1 (receive)
	3	14	86	0 (transmit)
		15	87	1 (receive)

On the SBC, the **init** routine is responsible for programming the interrupt vector number. Each successive controller is assigned interrupt vectors starting with the next multiple of 16. The next controller interrupt vector numbers start at 16, the interrupt vector numbers of the next controller start at 32 (regardless of the number of interrupt vectors assigned to the first controller), and so on. Refer to the **init** routine for a disk driver in Appendix E, lines 383 to 415 for an example of how the driver determines the proper interrupt vector to program into the board.

Absolute Assignment of Interrupt Vectors

Integral devices and devices whose interrupts are first processed by an intermediary interrupt handler (for example, SCSI devices) do not have direct entries in the EDT, and so cannot be assigned interrupt vector numbers in the same fashion as devices that do. These devices, such as the system console, must have their starting interrupt vector number hardcoded in the FLAG column of their driver's master file.

The following drivers support devices whose starting interrupt vector number can be entered in the FLAG column:

- drivers for integral devices
- software drivers
- drivers for SBC-VME devices with non-programmable interrupt vectors
- drivers that access extended bus devices such as SCSI

The starting vector number is then assigned to the interrupt vector table when the system boots.

Servicing Interrupts

When a user process issues an I/O request, such as a **read** or **write**, it must wait for the transfer to be completed, and so it uses the **sleep** function as discussed in Chapter 9. Similarly, an **open** routine may sleep until the device interrupts and announces its connection. When the device interrupts the CPU, the CPU calls the driver's interrupt routine. The driver interrupt routine then calls **wakeup** to inform the process that the transfer is complete.

The interrupt handler is responsible for identifying the reason for the interrupt (device connect, write acknowledge, data available) and set or clear device state bits as appropriate.

The following illustrates how the system handles operational interrupts:

1. A process accessing the base level of a driver issues an I/O request and goes to sleep awaiting its completion. The code that calls the **sleep**(D3X) function should be protected with **splhi** as discussed in Chapter 9. Going through the appropriate switch table[8], the I/O transfer is requested.

2. When the I/O transfer is complete, the I/O board requests an interrupt by sending a signal on the bus.

3. The CPU board receives the interrupt signal and passes it on to the microprocessor.

4. The interrupt acknowledge hardware determines which device is signaling the interrupt and accesses a table of interrupt vectors to transfer control to the appropriate driver's interrupt routine.

5. The driver's interrupt routine generates a **wakeup** call. The process that was suspended in the base level of the driver then sends the data to the user.

8. Switch tables are discussed in Chapter 2.

Writing Interrupt Routines

Interrupt routines are written for all hardware drivers that have interrupt capability. The device's controller must be physically attached to the bus of a computer to have an interrupt routine initiated by the CPU. Devices that reside external to the computer such as the SCSI bus which is attached to an external bus, do not generate interrupts in the same manner as internal devices. (The ABUS for the 3B4000 computer is considered an internal bus.)

The UNIX operating system defines three general names for the types of interrupt handling routines that must be written for UNIX devices: **int**(D2X), **rint**(D2X), and **xint**(D2X). If the device sends one interrupt, then the driver must include an **int** routine that uses **case** statements to determine the kind of interrupt that was sent. If the device sends two separate receive and transmit interrupts, then the CPU can determine the kind of interrupt being sent and the driver includes separate **rint** and **xint** routines for each type of interrupt. Descriptions of these routines found in the D2X section of the *Reference* guide.

In general, every interrupt routine must be responsible for the following tasks:

- keeping a record of interrupt occurrences
- interpreting the interrupt routine argument into a meaningful device or subdevice number
- rejecting requests for devices that are not served by the device's controller
- processing interrupts that happen without cause (called spurious interrupts)
- handling all possible device errors
- waking processes that are sleeping on the resolution of an interrupt request

Depending on how the master file information is stated when an interrupt occurs, either the **int** routine, or the **rint/xint** set is called. Interrupt routines for external devices can be named in any manner since they must be called by an intermediary driver (for example the host adapter driver for SCSI drivers). The names for these routines are conveyed to the system by special device structures. SCSI drivers, for example, inform the host adapter of the interrupt routine name via the **sc_int** member of the SCSI control block structure.

Writing an interrupt routine requires a merging of disciplines. As a driver developer, you must visualize the workings of the hardware and firmware to be able to write an effective interrupt routine. As already explained, an interrupt is generated by the hardware. For the purposes of writing your driver, you should know the exact chip set that produces the interrupt. You need to know the exact bit patterns of the device's control/status register and how data is transmitted into and out of your computer. This information differs for every device you access.

The Interrupt Routine Argument

To avoid having to create an interrupt routine for every possible interrupt vector, 3B computers developed a method of passing an argument to the interrupt routines. By passing an argument, one interrupt routine can handle many different interrupt vectors. However, not all interrupts receive or need parameters.[9]

The name of this argument to the **int**(D2X) and **rint**(D2X)/**xint**(D2X) routines, *ivec*, is slightly misleading, as its value is not the interrupt vector number associated with the interrupt. Rather, the *ivec* argument represents a "logical" interrupt number and its value is determined by the driver. Each driver may use *ivec* differently, depending on whether the board generates one interrupt vector per subdevice, one per controller, or some other arrangement.

The *ivec* argument can provide two important pieces of information to the driver. The first is the logical controller number. The logical controller number is the logical number of the controller supporting the device. This number is assigned by the system when the EDT is built. The second is the logical device number for the device causing the interrupt for that controller. A maximum of 16 logical interrupt numbers can be assigned per controller, one for each subdevice.

For example, if a controller supports one device, the logical interrupt value for the *ivec* argument represents the logical controller number. If a controller supports four subdevices and must send an interrupt for each, then the logical interrupt value for the *ivec* argument represents both the logical controller number and the logical device number of the device sending the interrupt.

ivec values begin at 0 and are incremented upwards. For example, for two controllers issuing four interrupts each, values 0 through 3 would represent controller 0 and its four subdevices. Values 4 through 7 would represent controller 1 and its four subdevices. The two tables presented in the "Interrupt Vector Number Assignment" section include *ivec* assignments for two sample configurations. See these tables for more examples of *ivec* assignments.

Interrupt Routine Restrictions

You must keep the following restrictions in mind when developing an interrupt routine:

- Interrupt routines must not set any fields in the `user` or `proc` structures, because the interrupted process is independent from the interrupt. For the same reason, interrupt routines must not call the **sleep** function directly or indirectly. The following functions either call **sleep** directly, or access the user or proc structures:

9. For the 3B2/3B15 passing of parameters to interrupt routines is done through the use of "assembly assist" routines. These assist routines are entered first from the interrupt Process Control Block (PCB) and then call the "real" interrupt routines. Some of these interrupt assist routines are "hard" coded in the operating system. The use of these assist routines also allows a common "return from interrupt" routine. This is very important for the UNIX operating system since at the end of every interrupt some system processing must be done. For the 3B systems which use "self-configuration" the driver assembly assist routines are built by self configuration.

canon	**getvec**	**sptfree**	**ttread**
copyin	**iomove**	**subyte**	**ttwrite**
copyout	**iowait**	**suser**	**ttywait**
delay	**kseg**	**suword**	**unkseg**
drv_rfile	**longjmp**	**ttclose**	**useracc**
fubyte	**physck**	**ttiocom**	
fuword	**sleep**	**ttioctl**	
geteblk	**sptalloc**	**ttopen**	

Table 10–2 Unavailable Interrupt Routine Functions (D3X)

- **spl*** functions must not drop the processor execution level below the level set for the interrupt routine. Doing so can corrupt the stack.

  For example, an integral disk drive (IDFC) on a 3B15 computer has an IPL value of 5 and the IPL bit in the Program Status Word (PSW) is set to a processor execution level of 10 (on the 3B15 computer, **spl6** is equivalent to a PSW IPL value of 10). If you set the processor execution level below **spl6**, then an interrupt from another device can take precedence over the IDFC interrupt and may corrupt the stack.[10]

10. Refer to the **spl*** manual page in Chapter 3 of the *BCI Reference Manual* for a table that relates the **spl*** function to the IPL values (**spl6** is for IPL 10 on the 3B15 Computer). See also "Preventing Interrupt Contention" in this chapter for more information on protecting critical sections of interrupt routines.

Writing Data Receive and Transmit Interrupt Routines

Transmit and receive interrupt routines must be written for character devices that send specific transmit and receive interrupts to the CPU. Because the two interrupts are unique, the CPU can determine which type of interrupt was sent, and so can associate the interrupt with a specific routine. Character drivers for these device require special interrupt routines to send data to a terminal and to receive data from it. The **rint/xint** routines are provided for this purpose.

Generally, a device that sends separate transmit and receive interrupts is not an intelligent device. An interrupt must be sent each time a character is transmitted or received. The following procedures outline **rint** and **xint** routines for unintelligent terminal devices that transmit and receive one character at a time.

Writing a Receive Interrupt Routine (rint)

When a character is received from a terminal device, a receive interrupt is sent to the CPU which associates the interrupt with the device's **rint**(D2X) routine. The **rint** input argument is used as an index to the device that generated the interrupt. This is not a device number as described by **dev_t**, but an integer value. When interfacing with a terminal, follow these steps:

1. Determine the subdevice number from the *ivec* argument to the **rint** routine.
2. Increment the interrupt-received flag. Commonly, the `sysinfo`(D4X) **rcvint** flag is used. (This long integer variable is defined in *sysinfo.h*.)
3. Check the control and status register (CSR). On terminal devices supported by AT&T 3B systems, the CSR is usually a structure associated with the Universal Asynchronous Receiver-Transmitter (UART). If the UART has a receive-ready status, continue with the next steps. Otherwise, exit the routine. (The proper UART is selected with the **rint** routine's input argument. All subsequent descriptions of UART access assume the appropriate UART has been selected.)
4. Reset the error status information register on the UART.
5. Read in a character from the UART. This is typically accomplished through a **while** loop that receives one character at a time as long as there are characters to receive.
6. If the terminal has start/stop control enabled, test the character to determine if it is a stop character (such as CTRL-s) or a start character (such as CTRL-q). To start the display, call the **proc**(D2X) routine with the T_RESUME flag set. To stop the display, call the **proc** routine with the T_SUSPEND flag set. After processing the character, exit the routine. If the character is not a start or stop character, continue.

7 Check the character for an error in framing or parity, for display overrun, and for being a BREAK character. Process according to the state of the `termio` structure's **c_iflag** member as explained in **termio**(7).

8 Read the character into your line buffer.

9 Echo the character back to the screen.

Writing a Transmit Interrupt Routine (xint)

When a character is ready to be transmitted to a device, the device driver's **xint** routine is called. Generally, the device is a terminal and access to the terminal is provided via a Universal Asynchronous Receiver-Transmitter (UART). Follow these steps for a transmit interrupt routine:

1 Determine the subdevice number using the *ivec* argument to the **xint** routine.

2 Increment the transmit-interrupt flag. Commonly, the `sysinfo.xmtint` flag is used. (This long integer variable is defined in *sysinfo.h*.)

3 Check the control/status register (CSR). On terminal devices, the CSR is usually a structure associated with the Universal Asynchronous Receiver Transmitter (UART). As long as the UART is showing a transmit-ready status, continue with the steps listed here. Otherwise, exit the routine. (The proper UART is selected with the **xint** routine's input argument. All subsequent descriptions of UART access assume the appropriate UART has been selected.)

4 While the CSR indicates a transmit-ready state, continue processing the interrupt. If this state is not evident, exit the routine.

5 Check the **t_state** member of the `tty`(D4X) structure. If the TTXON or TTXOFF flags are set (indicating that a start or stop character must be transmitted)

- transmit the proper characters to the terminal (via the UART)
- disable the respective flag in **t_state**
- exit the routine

6 Set **t_state** to BUSY and send the next character to the terminal.

Writing Interrupt Routines for Intelligent Boards

Intelligent boards provide the facility to share a queue with the interrupt handling routine and can take on some responsibility for moving data to and from the device. By using queues in memory, the number of interrupts that need to be requested by the device can be reduced. Devices controlled by unintelligent boards, frequently TTY devices, must interrupt the CPU each time a character is sent or received.

The driver's **init** or **start** routine formats an area of memory as a circular queue with pointers to the beginning and end of the queue. When this queue is set up, **init** notifies the board by writing a start-up message directly into the hardware. Typically, until the board has been successfully **sysgen**ed, the board waits for "stand-alone" commands sent by the driver that poll an area on its internal memory. The driver first formats a command buffer, then writes one word into the board memory to indicate that a command has been issued. That command contains pointers to the places in memory where the board should look for jobs that are associated with this device, such as the job request queue and the job completion queue. Typically, the driver writes a job in this buffer, updates the load pointer to indicate that there is a job waiting, and signals the hardware by either a control status request (CSR) bit or through some mechanism on the board that causes it to look at the job queue.

The advantage of this protocol is that it avoids memory contention between the hardware and the software because the driver updates the load pointer and the hardware updates the unload pointer when it gets the job. When the job is completed, the hardware puts a job in the queue (assuming there is room), updates the load pointer, and sends an interrupt to indicate that the job is completed. The driver interrupt routine checks the data structures to determine which of the devices interrupted and how many jobs are in the queue.

The following section discusses some specific concerns when sharing structures between a driver and a device.

Shared Driver/Device Structures

Structures shared between a driver a device present some specific difficulties that must be addressed by the interrupt routines.

- Information in the shared structure may be updated at any time by the device. The structure must be monitored frequently by the interrupt routine so that the structure is not abruptly changed. **spl*** functions cannot be used to prevent the device from changing a structure shared between a driver and hardware; only previously agreed on protocol can accomplish this task (where the hardware is smart enough to examine a flag in the control/status register to determine if it is safe to update the structure).
- Additional interrupts may occur signaling the placement of jobs on the request queue while the interrupt routine is processing a previous interrupt. One means of handling this problem is to have a loop that compares the load and the unload pointers on the

completion queue.

A job placed on the queue cannot be seen or acknowledged by the driver code when the driver is in the interrupt routine. What the driver can see is that the load pointer has moved. Using this indicator, the driver can handle the new job. This presents an additional problem: the driver interrupt routine must be prepared to unload more than one job from the queue.

- An interrupt is normally requested after the last request is processed. Since this interrupt is issued by the last request, the last job will have already been unloaded. This interrupt has no job associated with it and the interrupt routine must recognize that this interrupt is not an error condition.

  One way to ensure that the last interrupt is a holdover with no work attached to it is to keep a count of the number of jobs outstanding. The counter is incremented when the job is put on the request queue and decremented in the interrupt routine when the job is removed from the queue. Generally, this information may be kept in a separate data structure used for job status for each device or controller.

Figure 10-2 illustrates how a driver interrupt routine tests load and unload pointers. The interrupt routine shown in the example makes the following assumptions about the queue and the queue's load and unload pointers:

1 The completion queue contains two or more elements and is circular.

2 The queue is full when the load pointer plus one equals the unload pointer, and empty when the load pointer and unload pointers are equal.

3 The unload pointer always follows the load pointer.

4 Queue elements are loaded and unloaded consecutively.

5 The load pointer indicates where the next job will be placed; that is, the load pointer points to an empty element.

6 The load pointer is only updated by whatever fills in the elements.

7 The unload pointer indicates where the next completed element to remove resides.

8 The unload pointer is only updated by the interrupt routine.

9 The completion queue element(s) are filled in and the load pointer is updated before the interrupt is issued.

```
 1   drv_int(logical_dev)
 2   int logical_dev;   /* This is the logical device number */

 3     {
 4     struct drv *drvpt;  /* Pointer to the device
 5                 * structure.  Get the
 6                 * device structure for
 7                 * the logical device
 8                 * requesting service.  */

 9     drvpt = &drvstruct[logical_dev];

10               /* Check if work is pending
11                * by testing the load and
12                * unload pointers.  If they
13                * are equal, then there is
14                * no work to do.
15                */

16     if (drvpt->compq.loadptr == drvpt->compq.unloadptr)
17        return;       /* For some applications
18                * this may be an error condition
19                * that requires some action. */

20               /* Work pending, so
21                * unload queue until
22                * the pointers are equal
23                * More than one job
24                * can be unloaded.    */
```

Figure 10–1 Testing Interrupt Routine Load and Unload Pointers (*part 1 of 2*)

```
25      while (drvpt->compq.unloadptr != devpt->compq.loadptr)
26          {
27          unload job from completion queue;
28          perform necessary steps to
29              signal this job completed;
30          check for unload pointer going
31              past end of queue;
32          update unload pointer as required;
33      }

34                  /* All jobs have been
35                     removed, so exit   */

36      return;
37      }
```

Figure 10–1 Testing Interrupt Routine Load and Unload Pointers (*part 2 of 2*)

Writing int Interrupt Routines

An **int** routine is written for a device that sends one type of interrupt. The interrupt routine itself is responsible for determining the type of interrupt requested. Both character and block devices utilize intelligent controllers. The following sections provide examples of interrupt routines for both an intelligent character device and an intelligent block device.

Interrupt Routines for Character Devices

Some character devices send only one type of interrupt and are intelligent enough to share request and completion queues with the device driver. Interrupts are requested when a job is transmitted or received. Typically, a flag is set in the CSR by the device that determines what type of interrupt has been requested. The interrupt routine must use a **case** condition statement to provide separate sections of code to handle either case.

The interrupt routine for the driver illustrated in Appendix D (line 179) is an example of an **int** routine for an intelligent character device.

Interrupt Routines for Block Devices

Block devices are typically controlled by an intelligent controller that sends one type of interrupt. Block device interrupt routines must determine the reason the interrupt was requested.

The interrupt routine provided in Appendix E is an example of an **int** routine for an intelligent disk controller.

Preventing Interrupt Contention

Interrupts do not occur in isolation and in an orderly and coherent fashion. Interrupts from all the devices on the system can occur at any time and can impact both the base and interrupt portions of one driver, as well as two drivers sharing common data. If an interrupt switches control of the system from the base portion of a driver to the interrupt driven portion of a driver, the common data they are sharing may be corrupted by contending instructions.

When two sections of kernel code have a common interest in the same data, the driver must be able to coordinate access. Driver code that accesses common data is identified as a *critical section*. The word *section* refers to a portion of code that affects the common data, rather than the data itself. A *critical section* of code is one that manipulates data that is of concern to another piece of code capable of interrupting the first.

To get a clearer understanding of how interrupt contention can cause damage to common data, consider the following example:

A section of code in the base or synchronous portion of a hypothetical driver sets status flags as a way of communicating to the interrupt portion of the driver. Another section of code in the interrupt portion of the driver also sets those flags. Both sections of code do not set the flags in a single machine operation.

The synchronous portion of the driver receives a request that requires it to set the values of several flags. In the midst of setting the flags, the device requests an interrupt, transferring control to the interrupt portion of the driver. The condition of the interrupt forces the interrupt routine to first consult the current flag values set by the base portion of the driver, and then set them to new values.

Because the interrupt occurred before the base level portion of the driver could set the flags properly, the interrupt routine did not find the flags set to their proper values. Corruption like this could cause the interrupt routine to lose sanity, or it may simply continue the corruption. When the interrupt returns, the synchronous portion of the code, unaware that it was interrupted, finishes the changes it had started.

The section of code in the synchronous routine that shares data with the interrupt routine is the *critical section*. Whether the data identified in a critical section is changed by the interrupting routine is unimportant. The section is considered critical if a portion of code that manipulates data can be interrupted.

Critical sections of code must be protected from being interrupted when accessing critical data. The **spl***(D3X) functions permit code to set the processor's execution level so that interrupts are serviced in order of priority. When a critical section is identified, it can be protected from interruption by a call to an **spl*** function of the proper level. The following section discusses the use of these **spl*** functions.

Setting Processor Priority Levels

The system allows devices to interrupt the CPU and request immediate handling of interrupts. The integrity of system data structures could be destroyed if an interrupt routine were to affect the same data structures as a process already executing in the driver.

To prevent such problems, the system has special functions that set the processor execution level so that the CPU prohibits interrupts below certain levels. The functions are **spl***(D3X) where * ranges between 0 and 7, corresponding to the priority level that it has in the kernel. These priority levels are defined on the **spl***(D3X) reference page.

In most cases, the **spl*** function is given a variable to which it can pass the old priority level. Another function, **splx**, takes the value of that variable as an argument and resets the processor priority level to that value. The **splx** function is useful in cases where the processor priority level may have been raised already, but the driver does not know that it has been raised sufficiently to block out the proper level of interrupts. When the driver is ready to lower the priority level, it should return the priority level to its previous value.

The following code illustrates the use of the **spl*** and **splx** functions. The **spl*** functions first sets the processor priority level to 5, then saves the previous priority level in **s** (line 2). In line 6, the **splx** then resets the processor priority to the value saved by the **spl*** function in **s**.

```
register int s;
s = spl5();
while ((cp = getcb(&tp->t_rawq)) != NULL)
   putcf(cp);
tp->t_delct = 0;
splx(s);
```

Figure 10–2 Sample spl* and splx Function Calls

Contention conditions can occur if the code containing **sleep** functions is not protected by **spl*** functions. For example, the following code segment in the base level of a driver causes a process to sleep until the condition bit is cleared (by some other code) in the `driver.state` field:

```
driver.state |= condition;
while (driver.state & condition)
   sleep(&driver.state, PRIORITY);
```

The following code segment in the interrupt routine for that driver checks the condition bit to determine if a process should be awakened:

```
if (driver.state & condition)
{
    driver.state &= ~condition;
    wakeup(&driver.state);
}
```

Given the above examples, a process accessing the base level of the driver could check the condition bit, find it true, and call **sleep**. However, should an interrupt from another device occur after the condition has been cleared but before the base level portion of the driver called **sleep**, the interrupt routine would assume the process was asleep and call **wakeup**. By the time the interrupted process does call **sleep**, the **wakeup** call will have already been issued and another one may never come. By bracketing the calls to **sleep** with **spl*** function calls, the driver prevents the contention condition.

```
x=spl5();
driver.state |= condition;
while (driver.state & condition)
   sleep(&driver.state, PRIORITY);
splx(x);
```

The above example protects the code from all interrupts occurring at a priority level less than or equal to 5.

NOTE: **sleep** contains a call to **spl0** (**spl1** on the 3B15 and 3B4000 computers) that re-enables all interrupts while this process is sleeping.

10. Since processes could sleep on the address for several events, the **sleep** call is enclosed in the **while** loop, so that when awakened, the code will again check that the condition is indeed no longer true. This is one reason that it is recommended that processes sleep on different address values for different sleep reasons.

Do not set **spl*** functions that mask clock interrupts for long sections of code as this will make your system clock sluggish. Refer to the **spl*** manual page in Chapter 3 of the *BCI Driver Reference Manual* for more information on which **spl*** command to use to block interrupts for the different devices.

Chapter 11: Error Reporting

Contents

Introduction

One of the most important aspects of writing a device driver is the correct handling of errors. This chapter presents general guidelines and discusses how to implement the various error-handling facilities and signals. Driver code must handle any error condition, or the consequences may be severe. For instance, a stray interrupt should be a trivial event, but could panic the system if the driver is not prepared to handle it. The panic could cause data corruption and physically damage the system.

This chapter presents general guidelines and discusses how to implement the various error-handling facilities and signals. Chapter 13, "Testing and Debugging the Driver," discusses how to test for proper error handling.

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

- Write the error condition to a structure so the driver knows about it. Usually, at base level, the error is recorded in the **u.u_error** member of the `user`(D4X) structure. At the interrupt or base level, errors on block devices can be recorded in the **b_error** member of the `buf`(D4X) structure.

- Retry the process. The error may be a transient problem. Some hardware device boards have retry capabilities; let these boards do the retry. But if the error is software related, the driver must decide how many times to retry.

- Report the error to a system error log. If the error is severe, take the faulty hardware out of service to minimize the damage and keep the system running normally.

- Report the error to the system administrator, either by printing it on the system console, or by writing it to **putbuf** (to be reviewed with the **crash**(1M) utility).

- Send a signal to a user process.

- Panic the operating system.

Recording Error Messages in System Structures

Base-level driver errors should always be recorded to the **u.u_error** member in the `user` structure. This is where a driver function checks to see if an error has already been logged.

Block-access devices should record errors in two members of the `buf` structure. The **b_flags** member is set to B_ERROR, indicating an error has occurred, and the **b_error** member is set with the actual error code. The error code is written to the **u.u_error** member of the `user` structure when the **iowait**(D3X) function returns from **sleep**. When writing error codes, make sure the code describes the error and is meaningful. All other devices can mark base-level routine errors by writing the error code directly to the **u.u_error** member of the `user` structure. If your driver uses a private buffering scheme, set up error-handling members in the buffer header, as discussed in Chapter 6, "Input/Output Operations."

If the **strategy** routine finds an error in setting up the I/O, or if the device reports an error with an interrupt, the driver should set the following members of the `buf` structure.

b_flags
: should have the B_ERROR bit ORed in. The driver should not assign a value to **b_flags** because that may erase other bit patterns that the kernel relies on. The driver must never clear the **b_flags** member.

b_error
: should be set to an appropriate error value. Typical values are: EIO, for some physical I/O error, ENXIO, for attempting I/O on non-existent device, and EACCES, for attempting to access a device illegally. The kernel later sets **u.u_error** with the value of **b_error**, so any appropriate value for **u.u_error** could be set. Refer to Chapter 4, "Header Files and Data Structures," for more information on error codes used in drivers.

b_resid
: should be set to the number of bytes that have not been transmitted.

The **b_error** and **u.u_error** members accept any error code defined in Table 11-1.

Because error codes change from release to release, refer to the *Programmer's Reference Manual* for system-defined driver error codes.

Table 11-1 lists error codes used by drivers.

Table 11–1 Driver Error Codes

Error Value	Error Description	Use in these Driver Routines (D2X)
EAGAIN	kernel resources, such as memory, are not available at this time; cannot open device (device may be busy, or the system resource is not available).	**open, ioctl, read, write, strategy**
EFAULT	an invalid address has been passed as an argument; bad memory addressing error	**open, close, ioctl, read, write, strategy**
EINTR	when a process is sleeping above PZERO without PCATCH ORed to the sleep priority and a signal is received, **longjmp**(D3X) is called, control returns to user and EINTR is set in **u.u_error**.	**open, close, ioctl, read, write, strategy**
EINVAL	invalid argument passed to routine	**open, ioctl, read, write, strategy**
EIO	a device error occurred; a problem is detected in a device status register (the I/O request was valid, but an error occurred on the device)	**open, close, ioctl, read, write, strategy**
ENXIO	an attempt was made to access a device or subdevice that does not exist (one that is not configured); an attempt to perform an invalid I/O operation; an incorrect minor number was specified	**open, close, ioctl, read, write, strategy**
EPERM	a process attempting an operation did not have required super-user permission.	**open, ioctl**
EROFS	an attempt was made to write to, or to open a read-only device	**open**

IMPORTANT: Before officially installing the driver, be sure to remove any debugging code not enclosed in conditional compiler statements, as described in Chapter 13, "Testing and Debugging the Driver."

Table 11-2 lists error values that should be set in your code when functions return failure values.

Table 11–2 Error Codes Mapped to Function Return Values

Function	Return Value	Condition	Error Code
copyin	-1	Paging Fault	EFAULT
		Invalid user/stack area	
		Invalid address	
copyout	-1	Memory management fault	EFAULT
		Invalid user/stack area	
		Invalid address	
physck	0	Block does not exist	ENXIO
physio			EIO
		DMA error	EFAULT
suser	0	Current user not superuser	EPERM
useracc	0	User does not have access permission	EFAULT

The **b_error** and **u.u_error** members each hold only one error code at a time; if no error has been logged, the value is "0". Because a second error code will overwrite any previous value, the driver should test that the error member is blank before writing a new code. For a permanent record of errors encountered, write the error to the system error log.

Figure 11-1 illustrates how errors are written to the `user` structure.

```
if (useracc(u.u_base, u.u_count, B_WRITE) == 0)
{
        if (u.u_error == 0)
        u.u_error = EFAULT;
            return;
  }
```

Figure 11–1 Writing Error Code to user Structure

Figure 11-2 illustrates how errors are written to the `buf` structure.

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

Figure 11–2 Writing Error Code to buf Structure

Note that the B_ERROR is ORed into the **b_flags** member. The driver should not directly assign a value to **b_flags** because that may overwrite other bit patterns required by the kernel.

Sending Messages to the Console

Some driver errors should be sent to the system console, so that the system administrator can be alerted to the problem, and a hard-copy record can be made of error messages received. Sometimes, however, an important message will be lost because the printer was off-line or jammed when the message was sent. Furthermore, messages sent to the console, if numerous, can significantly slow system performance.

An alternative way to record errors is by using the **od** command of the **crash**(1M) utility, which can be used to access a message buffer called **putbuf**. This section explains how a driver writer can direct error messages to one or both of these destinations[1].

Using the cmn_err Function

The **cmn_err**(D3X) function can be used to write error messages to the system console, **putbuf**, or both.[2] Except for some block device error conditions (which use **print**(D2X) routines, explained below), the **cmn_err** is the main channel for reporting driver errors.

The **cmn_err** function takes three arguments. The first, *level*, specifies the severity of the error. The second, *format*, is the message itself, and the third, *args*, contains any variable data that must be sent along with the message.

```
cmn_err(level, "format", args);
```

The *level* may be any one of four pre-defined constants, listed below in order of severity.

CE_CONT is used to display information not associated with an error condition, or to continue another error message.

CE_NOTE errors do not require immediate attention but should be noted by system administrator.

CE_WARN errors are caused by resource exhaustion not detrimental to the operating system. For example, running out of file table entries.

CE_PANIC causes a system panic. The results of using this value are discussed more fully below, under the heading "Panicking the System."

1. On the 3B15 and 3B4000 computers, most driver error messages may also be sent to the system error log, providing another alternative to the system console.
2. Note that the **printf** kernel function should not be used on UNIX System V Release 3 and later systems.

The second argument to **cmn_err** is the actual string to be printed, enclosed in double quotes ("). To send a message to **putbuf** only, use an exclamation point (!) as the first character in the string. This is especially useful for debugging messages, since they can be viewed using **crash**(1M) and yet will not slow the system as much as messages to the console do. Send messages to the console and not **putbuf** by using a carat (^) as the first character in the string. Omit both of these characters to direct the message to both the console and **putbuf**.

The remainder of the second argument is the text to be printed, in the format of a **printf**(3S) style string. The **d**, **D**, **o**, **s**, and **x** conversion characters used by **printf** are available. Always include device information in the string printed to identify the driver involved. Also include the driver routine name issuing the **cmn_err** and the major and minor device numbers.

The **cmn_err** function ignores a length specification used with the conversion character. For instance, the code segment in Figure 11-3 sends a message that the open function has been called. The minor/major number of the device will be printed in hexadecimal because the "%x" conversion character is used. Because the function call is enclosed inside the `#if TEST - #endif` construct, this message will not be part of the final driver code.

```
    register struct device *rp;
     rp = xx_addr[(minor(dev) >> 4) & 0xf)];
#if TEST
     cmn_err(CE_NOTE, "xx_open function called - dev = 0x%x", dev);
#endif
```

Figure 11–3 Using cmn_err for Information

The **cmn_err** function automatically adds "\n" to all strings. If used, the "\n will print a blank line below the message.

The third argument *(args)* is reserved for the variable value or values to be printed with the string. In the example above, the device number (dev) is the third argument.

Recording Errors with logmsg

The **logmsg**(D3X) function is frequently used in with **cmn_err** to ensure that an error message is displayed and retained for further analysis. **logmsg**(D3X) is used to place an error message in the */usr/adm/errfile* error file, which is accessible by the **errpt**(1M) error report command. The message can be up to 256 characters long and must be enclosed in double quotes ("). **logmsg** provides a way to log errors outside the range of existing error types or when a console is not be available. (The number of characters in the string is determined by the EMSGSZ constant defined in *erec.h*.) Messages longer than 256 characters are truncated.

Writing a print Routine

Any driver that has a **strategy**(D2X) routine must also have a **print**(D2X) routine. This routine reports errors to the console that occur during I/O operations normally handled by the system buffering scheme. One such abnormal condition would occur when the device is out of space.

This routine prints literals from the kernel routine that describe the error. The routine you code should identify the device and subdevice. For example, Figure 11-4 lists the **print** routine from the IDFC disk controller driver on the 3B15 computer.

```
dfprint(dev, str)
register dev_t dev;
char *str;
{
   cmn_err(CD_WARN,"%s on IDFC(%d) drive 0%o", str, (dev>>8) & 0x7f, dev&0xff);
}
```

Figure 11-4 dfprint Routine from 3B15 IDFC Driver

Panicking the System

The **cmn_err**(D3X) function called with the *level* set to CE_PANIC is used to send an error message to the console and panic the system. A driver should panic the system only when the error condition stops the system from functioning, such as when the *root* device loses sanity. The code segment shown in Figure 11-5 halts the system when a bad disk volume table of contents (VTOC) is found on the *root* device. All messages using CE_PANIC should be written to both the console and the **putbuf** (by omitting the leading "!" or " ^ " from the message string). Any condition that could cause a system panic must also be recorded in the system error log.

```
register struct device *rp;
rp = xx_addr[(minor(dev) >> 4) & 0xf)];
if (rp->error == BADVTOC && dev == rootdev)
      cmn_err(CE_PANIC, "xx_open: Bad VTOC on root device");
```

Figure 11–5 Using cmn_err to Panic the System

Writing to the Error Log (3B15 and 3B4000 Computers)

Logged errors, error reports, and error messages are a critical part of analyzing system problems. Error reports can help you look back over a period of time to pinpoint hardware problems. Error messages provide up-to-the-minute notification of both hardware and software troubles. The UNIX system also records general errors and places them in a central system error log file, */usr/adm/errfile*. The contents of the file are collected in the following manner.

When the system enters multiuser state, the **errdemon**(1M) (a system error-logging daemon) is started. **errdemon** collects error records from the operating system by reading a special file and places the errors in a designated file. If a file is not specified when the daemon is activated, error records are written to */usr/adm/errfile*.

logstray(D3X) is a function used to record spurious system interrupts, also known as stray interrupts. This function helps the driver developer define an unusual error type. An error record header is built. After an error has been logged with **logstray**, the system administrator can produce a summary report or an overview of errors for a specific device. No analysis of the error records is done by **errdemon**; that responsibility is left to **errpt**(1M).

errpt(1M) processes data collected by **errdemon** and generates a report of the data. If no particular files are specified as **errpt** options, **errpt** uses */usr/adm/errfile* as the file to report on. (See the *System V Administrator's Reference Manual* for the complete list of **errpt** options.)

Another utility used to display errors is **errdump**(1M). Use the **errdump**(1M) command to display the error history file, which includes the contents of various system registers and the last five error messages received. The output of **errdump** may be sent to a line printer. The output can help to trace the cause of a system crash.

Logging Disk Errors

Disk defects are logged separately from the general error logging information. These errors can range from marginal to severe. If an disk error is severe, it will be logged in the disk error queue and the system error log.

When a disk defect message is logged, it usually means that the data stored in the bad block is damaged or lost, or that the disk may be unusable in its current state. The system administrator should take immediate steps to use the disk error information to map out these bad blocks and restore the data in full to the disk.

The disk defect management feature allows the system administrator to rewrite internal defect tables of a disk. If a disk supports this feature, any physical error that occurs on it is logged, enabling the administrator to identify areas of the disk that are becoming corrupt. In order for a disk device to use this feature, the driver writer must

- Ensure that the current operating system includes the *hde.o* object module.
- **#include** the *sys/hdelog.h* and *sys/hdeioctl.h* header files in the driver code.
- In the driver's **open**(D2X) or **init**(D2X) routine, initialize disk defect management tables either on a controlling sector of the disk or as a static table in the driver code using the **hdeeqd**(D3X) routine. **hdeeqd** also initializes the `hdedata`(D4X) structure which contains members that must be defined.
- Use the **hdelog**(D3X) routine to log errors in the driver's interrupt handler routine.

Initializing Hard Disk Error Logging

When a disk device is being opened for the first time (usually with a **mount**(2) system call), the driver **open**(D2X) or **init**(D2X) routine run during initialization must identify the device and set up controlling information (`hdedata` structure) about the device using the **hdeeqd**(D3X) function. This function is called once per device.

The **hdeeqd** function takes three arguments

`hdeeqd(`*dev*, *pdsno*, *edtyp*`)`

The first argument is the device number (composed of the external major and minor numbers). The second argument is a pointer to the table in the physical description (PD) sector. The third argument identifies the type of the device. (See the *BCI Driver Reference Manual* page for this function for valid device types.)

HDE Functions and Structures

The **hdelog**(D3X) and **hdeeqd**(D3X) functions, the `hdedata` structure and the HDE demon all play an important role in logging disk errors. Their interaction is summarized below.

- At boot time, **hdeeqd** initializes a `hdedata` structure for every disk in the system. A demon for the HDE driver should also be started at boot time. See the next section, "HDE Demon" for further information.
- At the same time, **hdeeqd** also initializes an error queue in kernel memory. The structure of the error file is defined in *hdelog.h*.
- When an error occurs, a retry is made. If the retry is unsuccessful, the driver provides **hdelog** with error information, and puts a new `hdedata` structure in the error queue. This error queue is a list of bad blocks that have not been remapped. It resides in the kernel and not on the disk. If a disk error is severe enough, it may also be sent to the system error log.
- While **hdelog** logs the error on the error queue, the HDE demon displays the error message on the console alerting the operator to the problem.
- After an error has been logged, the system administrator can use **hdelogger**(1M) (for IDFC and Lark™ II disks) or **shdelogger**(1M) (for SCSI disks) to format the log and print out reports on all known bad blocks. The information is printed to the terminal that executes the utility, not to the console.

After a number of errors have accumulated, the administrator may examine the error queue and determine if any of the entries should be fixed. To fix the disk, the administrator will use the **hdefix**(1M) (for IDFC and Lark II disks) or **shdefix**(1M) (for SCSI disks) utility to remap bad blocks. Remapping a bad block causes that block address to be written to a Manufacturer's Defect Table (MDT) on the disk. The disk physical description (PD) sector points to the MDT.

This mapping allows the administrator to make the defective physical disk tracks inaccessible to the system and maintain system integrity. (For more information on the **hdefix** and **shdefix**, see the *System Administrator's Reference Manual*.)

HDE Demon

At system boot time, the HDE driver usually initializes a demon (background program). This demon prints logged errors on the console. The demon is necessary for the following reasons.

It may happen that a disk is going bad and starts generating hundreds of bad block reports. If the HDE driver or another disk driver printed these error messages, the entire system would be dedicated to printing HDE error messages since drivers have a higher priority than other processes.

An administrator would have a difficult time fixing the bad blocks while the HDE driver monopolized the system, printing these messages. To prevent this from happening, the demon (a user process) is started when the system is booted. The demon sleeps until a bad block report is received by the HDE driver. The HDE driver wakes up the demon, which then prints the pertinent error information on the system console.

When the demon prints the error, the process runs at a user-level priority. The administrator's processes now get at least equal time with the demon (because they both are user processes) and may take corrective action.

EXAMPLE 1

In the following example, the information is kept on a controlling sector of the disk. To initialize disk defect management, the following steps are taken:

- Allocate a system buffer with **geteblk**(D3X) (line 48). The disk defect table is created in this buffer, then written to the appropriate area of the disk.
- Read the controlling sector from the **xx_strategy** routine using the **iowait**(D3X) function (lines 53–54).
    - If an error occurred on the read attempt, it displays an error message using the driver's **print**(D2X) routine and returns an error condition (lines 55–58).
    - Otherwise, move information from the buffer to the controlling sector with the **bcopy**(D3X) function (line 60), initiate error logging for the device with **hdeeqd** (line 61), and indicate that the device has been opened (line 62).
- Release the system buffer with the **brelse**(D3X) function (line 64)

```
1   #define XX_CNTLBLKNO  0  /* Block number of controlling sector */

2   struct  device          /* Layout of physical device registers */
3   {
4           char    reserve[4]; /* Reserve space on card          */
5           ushort  control;   /* Physical device control word    */
6           char    status;    /* Physical device status word     */
7           char    ivec_num;  /* Device interrupt vector number  */
8                     /* in 0xf0; subdevice reporting in 0x0f    */
9           paddr_t addr;   /* Address of data to be read/written */
10          int     count;  /* Amount of data to be read/written  */
11  }; /* end device */

12  struct xx_              /* Logical device structure */
13  {
14          struct buf  *xx_head; /* I/O buffer queue pointer head */
15          struct buf  *xx_tail; /* I/O buffer queue pointer tail */
16          short       xx_flag; /* Logical status flag          */
17          struct hdedata xx_edata; /* Hard disk error record log  */
18          struct iostat  xx_stat; /* Unit I/O statistics for     */
19        /* establishing an error rate during error logging      */
20  }; /* end xx_ */

21  struct xx_info              /* Information on control sector */
22  {
23      long    xx_id;          /* of disk device id code        */
24      long    xx_cyl;         /* Total number of cylinders     */
25      long    xx_trk;         /* Number of tracks per cylinder */
26      long    xx_sec;         /* Number of sectors per track   */
27      char    xx_serial[12];  /* Device serial number          */
28  }; /* end xx_info */

29  extern struct xx_    xx_devtab[]; /* Logical device structure table */
30  extern struct device *xx_addr[];  /* Physical device registers location */
31  extern struct xx_info xx_info[];  /* Device control information */
32  extern int           xx_cnt;     /* Number of devices          */
33   ...
```

Figure 11–6 Hard Disk Error Logging Is Initialized *(part 1 of 2)*

```
34  xx_open(dev, flag)
35  dev_t   dev;
36  int flag;
37  {
38  register struct xx_     *dp;
39  register struct device  *rp;
40  register int unit;
41   ...
42  unit = minor(dev) >> 4;         /* Get drive unit number            */
43  dp   = &xx_devtab[unit];        /* Get logical device information   */
44  if  ((dp->xx_flag & XX_OPEN) == 0)  /* First time opening the device,*/
45  {
46      register struct buf *bp;
47      hdeeqd(dev, XX_CNTLBLKNO, EQD_ID); /* Initialize error logging */
48      bp = geteblk();              /* Get a buffer for control sector */
49      bp->b_flags = B_READ;        /* Set up buffer to read           */
50      bp->b_blkno = XX_CNTLBLKNO;     /* Control sector from disk     */
51      bp->b_count = 512;
52      bp->b_dev = dev & ( 0xf);       /* Use partition 0 on disk      */
53      xx_strategy(bp);                /* Read control sector          */
54      iowait(bp);                     /* Wait for read to complete    */
55      if  ((bp->b_flags & B_ERROR) != 0 ) /* If data error occurred, */
56      {                               /* display message on console   */
57          xx_print(dev, "xx_open: cannot read control sector");
58          u.u_error = bp->b_error;    /* Get error code               */
59      } else {              /* Copy control sector data to info table */
60          bcopy(bp->b_un.b_addr, &xx_info[unit], sizeof(struct xx_info));
61          hdeeqd(dev, XX_CNTLBLKNO, EQD_ID); /* Initiate error logging */
62          dp->flag |= XX_OPEN;                /* Indicate device open   */
63      } /* endif */
64      brelse(bp);         /* Release system buffer */
65  } /* endif */

66  if  (u.u_error != 0)  /* If error found at this point, return */
67      return;
68  /* endif */
```

Figure 11–6 Hard Disk Error Logging Is Initialized *(part 2 of 2)*

EXAMPLE 2

A driver interrupt routine is responsible for checking for data transfer errors (these errors are called data checks). When a data check occurs (reported by the device in the status or error register), the driver determines if there have been sufficient attempts at resolving the error. If so, the driver abandons the I/O request by marking the buffer as being in error, logging an unresolved error with **hdelog**, and marking the I/O operation complete with **iodone**(D3X). When an error persists in spite of multiple attempts to resolve it, the driver logs marginal errors with **hdelog** and attempts the I/O operation again. NOTE: the driver may try to resolve the error with software by using the error correction bits in the error correction code (ECC) register.

```
1   struct  device          /* Layout of physical device registers */
2   {
3           char      reserve[4]; /* Reserve space on card          */
4           ushort    control;    /* Physical device control word   */
5           char      status;     /* Physical device status word    */
6           char      ivec_num;   /* Device interrupt vector number */
7                       /* in 0xf0; subdevice reporting in 0x0f    */
8           paddr_t   addr;       /* Address of data read/written   */
9           int       count;      /* Amount of data read/written    */
10  }; /* end device */

11  struct xx_                    /* Logical device structure       */
12  {
13          struct buf    *xx_head; /* I/O buffer queue head pointer */
14          struct buf    *xx_tail; /* I/O buffer queue tail pointer */
15          short         xx_flag; /* Logical status flag           */
16          struct hdedata xx_edata; /* Hard disk error record log   */
17          struct iostat  xx_stat; /* Unit I/O statistics for       */
18          /* establishing an error rate during error logging      */
19  }; /* end xx_ */
```

Figure 11–7 hdelog — Logs Media Errors *(part 1 of 3)*

```
20  struct xx_info        /* Information on control sector of disk   */
21  {
22           long    xx_id;                          /* Device id code   */
23           long    xx_cyl;           /* Total number of cylinders    */
24           long    xx_trk;          /* Number of tracks per cylinder  */
25           long    xx_sec;           /* Number of sectors per track  */
26           char    xx_serial[12];    /* Device serial number         */
27  }; /* end xx_info */
28  extern struct xx_     xx_devtab[]; /* Logical device structure table */
29  extern struct device *xx_addr[];   /* Physical device register location */
30  extern struct xx_info xx_info[];   /* Device control information */
31  extern int            xx_cnt;      /* Number of devices          */
32  ...
33  xx_int(board)
34  int  board;
35  {
36  register struct device  *rp = xx_addr[board]; /* Get device registers */
37  register struct xx_     *dp;
38  register struct buf     *bp;
39  register int    unit;

40  unit = (board << 4) | (rp->ivec_num & 0xf); /* Construct unit number */
41  dp = &xx_devtab[unit];

42  if  ((rp->status & DATACHK) != 0) /* If data check error occurred, */
43  {
44      if  (++dp->xx_edata.badrtcnt > XX_MAXTRY)    /* If sufficient  */
45      {   /* attempts have been made, then abandon the I/O request   */
46          bp = dp->xx_head;             /* Get buffer from I/O queue */
47          dp->xx_head = bp->av_forw; /* Remove buffer from I/O queue */
48          bp->b_flags |= B_ERROR;       /* Mark buffer in error      */
49          bp->b_error = EIO;            /* Supply error condition    */
50              /* Supply information needed for error logging         */
51          dp->xx_edata.diskdev = bp->b_dev;   /* The device number   */
52          dp->xx_edata.blkaddr = bp->b_blkno; /* The error block number */
```

Figure 11–7 hdelog — Logs Media Errors *(part 2 of 3)*

```
53          dp->xx_edata.readtype = HDEECC; /* Error type: error check */
54          dp->xx_edata.severity = HDEUNRD;    /* Data unreadable     */
55          dp->xx_edata.bitwidth = 0;
56          dp->xx_edata.timestmp = time;   /* Time recording occurred */
57          bcopy(dp->xx_edata.dskserno, xx_info[unit].serial, 12);
58          hdelog(&dp->xx_edata);  /* Log abandoned I/O operations    */
59          iodone(bp);             /* Mark I/O operation complete     */

60      } else if(dp->xx_edata.badrtcnt > 1) { /* If more then one retry,  */
61                                          /* log error as marginal       */
62          bp = dp->xx_head; /* Get buffer from I/O queue but leave on I/O */
63                              /* queue so I/O operation is repeated    */
64                          /* Supply information needed for error logging  */
65              dp->xx_edata.diskdev = bp->b_dev;   /* The device number   */
66              dp->xx_edata.blkaddr = bp->b_blkno; /* The block number in error*/
67              dp->xx_edata.readtype = HDEECC; /* Error type: error check */
68              dp->xx_edata.severity = HDEMARG;    /* Marginal error      */
69              dp->xx_edata.bitwidth = 0;
70              dp->xx_edata.timestmp = time;   /* Time recording occurred */
71              bcopy(dp->xx_edata.dskserno, xx_info[unit].serial, 12);
72              hdelog(&dp->xx_edata);          /* Log data check error    */
73  } /* endif */
74      } /* endif */
75   ...
```

Figure 11–7 hdelog — Logs Media Errors *(part 3 of 3)*

Signals

A signal is a type of message sent to user processes alerting them to an important event. Drivers send signals to user processes to alert them of conditions on the device. For example, when a user on a terminal presses the BREAK key, it generates an interrupt. When the terminal driver handles that interrupt, it sends a signal to any user processes in the process group for that terminal.

Signals are used principally by character-access drivers.

Sending a Signal

Signals are sent from a driver's interrupt handler or base routines to a user process with the **psignal**(D3X) and **signal**(D3X) functions. The **psignal** function sends a signal to a single process, whereas the **signal** function alerts a process group. The needs of the individual device determine the sorts of signals that are used. **psignal** usually sends a signal to the **u.u_procp** member of the `user` structure, but not from the interrupt level. **signal** usually sends a signal to the **t_pgrp** member of the `tty` structure. The user process can intercept the signal with the **signal**(2) system call.

Figure 11-8 contains example signal code.

```
62  if (code == L_BREAK) {
63          signal(tp->t_pgrp, SIGINT);
64          ttyflush(tp, (FREAD|FWRITE));
65          return;
```

Figure 11–8 Signal Code

A driver that sends signals must **#include** the *sys/signal.h* header file, which defines all available signals. Signals frequently used in drivers include SIGINT, SIGQUIT, and SIGHUP. Figure 11-9 illustrates signal handling.

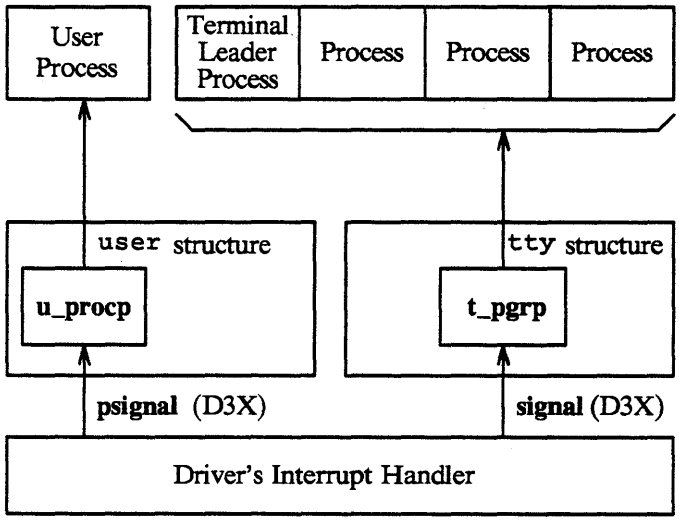


Figure 11–9 Processing Signals

Controlling Signal Priorities

The **sleep** function causes the current running process to sleep. The priority argument to the **sleep**(D3X) function determines if the user process will be awakened by signals or not. This is done in relation to the system-defined constant, PZERO (see Figure 11-10). Processes sleeping with priority values lower than or equal to PZERO will not be awakened by a signal; processes sleeping with priority values greater than PZERO will interrupt the current sleep and return to user level. (For more information on **sleep**, see Chapter 9, "Synchronizing Hardware and Software Events.")

Sleep Priorities 1–25	PZERO	Sleep Priorities 26-39
not awakened by signals	25	awakened by signals

Figure 11–10 sleep Priorities

You can use an absolute value (for instance, 27) as the **sleep** priority, but the preferred method is to use a value relative to PZERO (for instance, PZERO+2).

If the operating system handles the error processing, it simply returns the EINTR error code to the user program that called the driver. While EINTR is not very precise, the user program can use it as an indicator of a signal arrival. Generally, when EINTR is received at user level, the user program should retry the original command.

To process the signal in your driver, use the C programming language OR (|) instruction to add the value PCATCH to the priority argument that you assign for **sleep**, for example:

```
    if (sleep(&sleepaddr,(PZERO+1)|PCATCH)) {
            u.u_error = EINTR;
            cmn_err(CE_CONT,"Disk drive #103 is getting flaky");
            return;
}
```

Figure 11–11 sleep and PCATCH

Should a signal be received by a call to **sleep** with the priority OR-ed with PCATCH, **sleep** returns a value of 1 (true).

NOTE: Being awakened from a **sleep** call does not end the life of a signal. The user-level program should have invoked a mechanism for trapping signals that can provide further insight into what may have caused the error.

Chapter 12: Installation

Contents

Introduction

Installing a driver, also called configuration, consists of creating or modifying a series of files to ultimately produce a bootable object file. Then when the computer on which you are working is shutdown and brought up again, a new version of the operating system is created that includes your driver as part of the kernel.

This chapter provides the following information:

- installing a driver for the first time
- installing an existing driver
- suggestions for installing the driver during the testing/debugging phase of development
- installation of a driver when you are using a different type of computer for development than the computer for which the driver is being written (cross-environment)
- how to remove an installed driver from the computer

If you are installing your driver on several computers or selling it to other customers, create INSTALL and UNINSTALL scripts that run through the **sysadm**(1M) administrator command. This and other concerns when packaging a driver are discussed in Chapter 15.

This chapter tells you how and when to create or modify the files used for self-configuration and system initialization.

Installing a Driver For the First Time

When installing your driver for the first time:

- Create a master file
- Create special device files
- For a software driver, insert a line in the */etc/system* file
- For a hardware driver on the 3B2 computer or SBC, create the diagnostics files
- For a hardware driver on the 3B2 computer or SBC, add the device to the EDT
- For a hardware driver on the 3B2 computer or SBC, move pump files to a special directory

This section contains information that precedes subsequent sections in this chapter. If you have already installed a driver using the material described in this chapter, precede to the next section, "Installing an Existing Driver" for information on how to install your driver on a specific computer.

NOTE: You can install your driver software from any directory except */boot*. */boot* is not usable because an object file created by the **cc** command stored in */boot* may prevent a new operating system from being generated. After you have completed installing a driver and have tested it, you may wish to move the source and object code to the */usr/src/uts/<computer-type>/io* directory, making new directories as required. This directory typically contains driver source code. (*<computer-type>* choices are explained later.) 3B4000 computer adjunct processor code should be stored in the */usr/add-on*/package-name/*io* directory, once again, you should create directories as needed.

Creating a Master File

The easiest way to create a master file is to copy an existing master file; this saves time because you do not have input column headers. Master files reside in the */etc/master.d* directory or in the */adj/pe#/etc/master.d* directory for a 3B4000 adjunct processor. Each file is named for the driver it defines, in lower case letters, and corresponds to a file in the */boot* directory (*/adj/pe#/boot* for an adjunct processor) that has the same name in upper case letters.

The master file fields are separated by either a tab or a blank; no field can contain a blank. Any line with an asterisk (*) in column 1 is treated as a comment. By convention, each master file begins with a comment line that has the name of the driver, followed by another comment line that gives column headers for the fields used in the file. The fields in the second comment line define the configuration information for the driver.

The following is an example of the *console* master file. The following master file is used as an example throughout this section:

```
* console
*
*FLAG    #VEC    PREFIX    SOFT    #DEV    IPL   DEPENDENCIES/VARIABLES
orcst24  1       con       0       1       7
                                                 con_tty[2] (%0x58)
                                                 con_cnt(%i) = {2}
```

Figure 12–1 Console Driver Master File

Master File Fields

Each field in the master file contains configuration information specific to your driver. Some fields can be filled before you begin development, such as the FLAG, PREFIX, and SOFT columns. Others may only be filled once the restrictions placed on your driver by the device hardware have been determined, such as the #VEC, #DEV, and IPL columns. The DEPENDENCIES/VARIABLES information cannot be included until the dependencies of your driver on other drivers and/or defined structures, and the number and types of variables needed for your driver have been determined.

The following sections discuss the contents of each field.

FLAG

The FLAG field of the master file contains a combination of letters and/or numbers defining a number of characteristics specific to your driver for the system's boot programs. Each letter and number indicates a specific characteristic of the driver. The following list describes each symbol:

Access and Interface Definers

b Indicates that the driver's device supports block-access. This letter must be included if the driver includes a **strategy**(D2X) routine.

c Indicates that the driver's device supports character-access. This letter must be included if the driver includes a **read**(D2X), **write**(D2X), or **ioctl**(D2X) routine.

f Indicates that the driver is a STREAMS driver (both hardware and software).

m Indicates that the master file is for a STREAMS module.

s Indicates that the driver is a software driver. If the **s** flag is used, the **drvinstall**(1M) command will put the major number in the SOFT column.

t Indicates that the device uses the `tty` structure. This flag causes the **cdevsw[].d_ttys** field to be initialized for the device.

x Indicates that the master file is for a loadable module that is not a driver.

Other Configuration Instructions

o Indicates that only one device can be configured for this driver.

r Indicates that this device must be present or the system should not be configured. For instance, the *console* and *mem* (memory) master files use this flag.

a Indicates that **lboot** should generate and fill a segment descriptor array. The name of this array is:

extern paddr_t *prefix*_**addr**

number The first interrupt vector for an integral device. For an SBC with a non-programmable interrupt vector, the interrupt vector physically set on the board (either with DIP switches or with connectors) must be specified in this field in decimal.

In the *console* master file example, the following characters and numbers are used:

```
orcst24
```

This indicates the following about the driver:

o Only one device can be configured for the driver.

r The device supported by this driver must be present in order to configure the system.

c The device supported by the driver is a character device. See Chapter 6 for more information on block and character access.

s The driver is a software driver.

t The device is a TTY device and the driver uses the `tty` structure.

24 The first interrupt vector for the device is assigned to be 24. Software drivers can have their interrupt vector permanently assigned. See the #VEC section and Chapter 10 for more information on interrupt vectors and absolute address assignment.

#VEC

The #VEC column defines the number of interrupt vectors to be generated for each device or device controller. An interrupt vector is an offset to an interrupt vector table the system uses to associate interrupts with their appropriate interrupt routines, and with their appropriate devices.

The number of interrupt vectors a device needs is dependent upon how the device initially sends its interrupts. For instance, a controller that supports four subdevices may interpret those interrupts itself, or it may not. If it does interpret them, only 1 interrupt vector must be assigned to that device, and the controller determines the type of interrupt being sent. If it does not, 4 interrupt vectors must be assigned to the device, one for each subdevice.

In the *console* driver example above, 7 interrupt vectors are supplied.

The #VEC field in the master file defines the number of interrupt vectors per device, in this case per controller:

One interrupt vector per device (controller)
If the value of #VEC is 1, the controller itself has only one interrupt vector. Either the device supported by the driver does not support subdevices or the driver must determine which subdevice is associated with a given interrupt in some other way, such as by reading a controller register. Most intelligent controllers on the 3B15 and 3B4000 computers use completion queues rather than vectors, so use #VEC=1.

One interrupt vector per subdevice
If each subdevice has one interrupt vector and the controller can support up to four subdevices, #VEC is assigned a value of 4.

Multiple interrupt vectors per subdevice

Some character-access subdevices require more than one interrupt vector. For example, a serial port that has separate receive and transmit interrupts (coded using the **rint/xint** combination) must have two interrupt vectors per subdevice. If the sample configuration is for such devices, the value of #VEC is 8.

Refer to Chapter 10 for information on the handling and the assignment of interrupt vectors.

PREFIX

The 2-, 3-, or 4-digit prefix assigned to your driver and used as a prefix to the system routines. The kernel uses the driver's prefix to identify the appropriate kernel routine to use for this driver. The most important thing to remember about driver prefixes is that they must be unique. Different drivers cannot use the same prefix or their routines would be mismatched. Ensure that the prefix you select is unique by examining all other master files.

SOFT

The SOFT column is used to identify the major number for a software device[1]. Software device major numbers can either be automatically assigned by the **drvinstall**(1M) command, or hardcoded by the driver writer. If you wish to have the **drvinstall** command assign the major number, enter a dash (-) in this column. Master files for drivers supporting hardware devices should contain only a dash. See "determining Major and Minor Numbers" for more information on major number assignment.

#DEV

The #DEV column defines the maximum number of subdevices the device controlled by this driver can support.

IPL

The IPL column defines the interrupt priority level (1 to 15) at which the processor's CPU will service the interrupt request. Level 0 is the highest priority and level 14 is the lowest. Level 15 indicates that no interrupts are waiting to be serviced.

The CPU services interrupts based on its current processor execution level and in order of interrupt priority. The interrupt's IPL is the priority level at which the interrupt is requesting service. The CPU's processor execution level is the level at which the processor is executing. If the IPL is a higher priority than the current execution level, the CPU stops it's current execution, sets its execution level to the level of the IPL, and services the interrupt. If the IPL is a lower priority than the current execution level, it is queued until the CPU services those interrupts with higher priority.

1. The master file for a software device contains an "s" in the FLAG column.

A device's interrupt priority level is usually strapped in hardware and is totally independent of slots or interrupt vectors.. The interrupt request level for a device is marked by one of the bergs (physical connectors) on the backplane. The IPL value to use in the IPL column of the master file is usually included in the installation documentation for the device.

However, a device's IPL value can be overridden for critical sections of code with the **spl***(D3X) function. See Chapter 10 and the **spl*** manual page for more information on **spl*** function and setting interrupt priority levels.

DEPENDENCIES/VARIABLES

The DEPENDENCIES/VARIABLES field can have several lines. This field is used to

- Define other driver(s) on which this driver is dependent. (A driver is considered dependent on another if by the lack of the other driver, the former will not work.) For example, for two drivers X and Y, if */etc/system* has INCLUDE X and the */etc/master.d/A* has "B" in the Dependencies field, **lboot** will bring in X (based on */etc/system*) and Y (based on the dependency).
- Generate dummy functions if driver is not loaded when the system is booted.
- Assign values to variables according to the capacity of the driver rather than the actual hardware configuration.
- Assign values to variables according to administrator-supplied information about the specific configuration.

Generating Dummy Routines

A dummy, or stub routine is simply a function call with no arguments and no instructions. An example is:

```
myroutine(){}
```

A stub routine allows the system boot program to resolve symbols when a driver is not included in the system. Other means for generating stub routines are shown in Figure 12-2.

Value	Description
{nosys}	Send SIGSYS to current process when accessed
{nodev}	Return ENODEV error code when accessed
{false}	Return 0
{true}	Return 1
{ }	No return value

Figure 12–2 Dummy (Stub) Routine Names

Variables Set for A Driver

Variable definition lines define certain variables to be calculated by the system at boot time. The line has four fields, two of which are optional, separated by specific field delimiters; the line can contain spaces as long as they are not between elements of the length specifiers. The format of a variable definition line is:

variable-name[*array-size*](*length*)={*elements*}

The *variable-name* and *length* fields are required. The *variable-name* corresponds to the name used in the header file (or global data structure declaration section) for the driver. The *length* specifies the length of the variable value with any combination of the following length specifiers:

%i	integer
%l	long integer
%s	short integer
%*n***c**	character string *n* bytes long (default = 1)
$n	field *n* bytes long

Each specification is properly aligned and the variable length is rounded up to the next word boundary during processing.

The *array-size* field specifies the size of the segment descriptor array to be generated. If you use the **a** flag under the FLAG column, you must use this field; otherwise you must not use this field.

The *elements* field is an optional field used to initialize individual elements of a variable. If the calculations are based on numbers which the administrator can tune according to the configuration, this field should be filled as described in the next section.

The *array-size* and *elements* fields are infix expressions. An infix expression is in the form of a standard equation such as 1 + 2 = 3.

Tunable Variables

Variables that will be modified by the system administrator should be defined using a tunable variable table at the end of the master file. To set this up:

1. Use a three to six character upper case string for the *elements* field in the variable definition line. For example:

```
err_neslot (%i) ={NESLOT}
```

2. After all DEPENDENCIES/VARIABLE, start the tunable table with the string "$$$" beginning in the first column of the row.

3. List each tunable on a separate line followed by a space, an equal sign, and the default value. For example:

```
NESLOT = 50
```

4. To change the value of the variable, the administrator will modify the value in the tunable table. Comment lines in the tunable table should give guidelines on setting the value.

Note that other variable definition line can use this tunable as they could use any other *elements*.

NOTE: If you are installing a software driver and have created an alternate master file directory, a risk exists that a duplicated major number may be assigned for a driver. Before installing a driver in the kernel that may have been previously assigned a major number, ensure that the number is unique before continuing (use the **grep**(1) command). If a number is duplicated, either use **grep** to find a new, unused major number, or edit the master file of one of the drivers to put a dash under the SOFT column and reinstall the driver using **drvinstall**(1M).

Creating Special Device Files

The special device files provide user level access to a driver. After a driver is installed, a user program accesses the driver by opening the special device file.

On the SBC and the 3B2 computer, special device files are created with the **mknod(1M)** command.

On the 3B4000 adjunct processors, special device files are created on the master processor with **mknod** only for testing purposes. When the device files are created as part of the system, the information is added to a special file called a *prototype file*[2]. This file contains a list of all the devices in use by an adjunct processor. The prototype file ensures that the device files are created thereafter each time the adjunct processor is put into service (booted).

The format for **mknod** is:

For character devices:

mknod *name* **c** *major-number minor-number*

For block devices:

mknod *name* **b** *major-number minor-number*

The first argument to the **mknod** command is the name of the special device file. The names of special device files have no meaning to the operating system itself, but some programs expect a particular name to reference a particular device.

The second argument is **b** for a block device or **c** for a character device. The third argument is the major number; the fourth argument is the minor number. (Refer to Chapter 3 for information on determining major and minor numbers).

As an example, use this command to create a character special file named */dev/grz01* with major number 32 and minor number 1:

mknod	**/dev/grz01**	**c**	**32**	**1**
	filename	*character device*	*major number*	*minor number*

A special device file can be removed with the **rm**(1) command; to modify a special device file, delete it with **rm** then recreate it.

2. Refer to "Adding to a Prototype File" at the end of this section for more information.

NOTE: On the 3B2 computer, the 3B4000 ACP, and the SBC, many devices have subdevices for which device files must also be created. If this is the case, use the instructions that follow. If not, move to the next subsection.

Use **edittbl**(1M) to check the subdevices for your device in the */dgn/etc_data* file. If the **subdev_name** field contains **Hard** or **Serial**, skip this step (the **/etc/disk**(1M) or **/etc/ports**(8) commands that are already set to run will create the appropriate special device files). Otherwise, create a shell script in the appropriate directory (*/etc/brc.d* or */etc/rc.d*) to generate special device files for the subdevices associated with your driver. Do this so that the */dev* files can be dynamically created at boot time to accommodate configuration changes. You may also want to add to the */etc/bcheckrc* shell script if your driver application will need to check file systems, date, or perform other activities before a file system is mounted.

Types of Special Device Files and Device File Names

The types of device files you create for the device depends on the kind of access your device supports. For instance, all terminals are character devices, and so require only character special device files. Disk devices, on the other hand, support both character and block access, and so require both character and block special device files. The following sections discuss the types of device files required for some commonly supported devices.

Tape Subsystem

A tape drive can be accessed as either a character (raw) device or a block device. The special files for tape are in the */dev/mt* directory (for block tape devices) and in the */dev/rmt* directory (for raw tape devices). Every tape drive has two entries in both directories, so any tape can be accessed as either a block or a raw device, with or without rewind. A tape drive with rewind automatically rewinds after the operation. You must make four new */dev* entries for each tape drive, using either the **sysadm**(1) **mkdevmt** or the **mknod**(1M) command. Each tape drive also has a file in */dev/SA* and */dev/rSA*; these are used by System Administration to access tapes, and are created with the **sysadm mkdevdsk** command.

NOTE: **sysadm** only recognizes existing devices. Use **mknod** to create */dev* files for new devices.

One convention is that the name of a tape special file with rewind is the tape drive number followed by an "l" (low density) for an 800 bpi (bits per inch) drive, "m" (medium density) for a 1600 bpi drive, and "h" (high density) for a 6250 bpi drive. For example, the special file for tape drive 0 is *0m* if it is 1600 bpi and *0h* if it is 6250 bpi. The name for using a tape special file without rewind is the tape drive number followed by *mn* for a 1600 bpi drive, and *hn* for a 6250 bpi (high density) drive. For example, the special file for tape drive 0 with no rewind is *0mn* or *0hn*. Tape drives without rewind enable you to write more than one file to one tape.

The minor number for a tape special file is calculated to indicate the type of access.

The traditional naming conventions and formulae for calculating minor numbers for tape devices are summarized in Figure 12-3. This is only valid for the AT&T tape driver. Another method that is in wider acceptance, particularly in SCSI products is described after the table. The question mark (?) represents the tape drive number.

Type	Special File	Minor Number =
Block, rewind; 1600 bpi	*/dev/mt/?m*	(4 * ?)
Block, no rewind; 1600 bpi	*/dev/mt/?mn*	(4 * ?) + 1
Raw, rewind; 1600 bpi	*/dev/rmt/?m*	(4 * ?) + 2
Raw, no rewind; 1600 bpi	*/dev/rmt/?mn*	(4 * ?) + 3
Block, rewind; 6250 bpi	*/dev/mt/?h*	[(4 * ?)] + 128
Block, no rewind; 6250 bpi	*/dev/mt/?hn*	[(4 * ?) + 1] + 128
Raw, rewind; 6250 bpi	*/dev/rmt/?h*	[(4 * ?) + 2] + 128
Raw, no rewind; 6250 bpi	*/dev/rmt/?hn*	[(4 * ?) + 3] + 128

Figure 12–3 3B15 or 3B4000 MP Minor Numbers and Names for Tape Devices

For example, the special file *Omn* in the *rmt* directory has the minor number 3, calculated from:

(4 * 0) + 3

The special file for the same device in the *mt* directory has the minor number 1, calculated from:

(4 * 0) + 1

Simple Administration accesses tape devices through special files in the */dev/SA* and */dev/rSA* directories. These files are linked to the appropriate files in the */dev/mt* and */dev/rmt* directories, and named */dev/SA/9track#* or */dev/rSA/9track#*, where # corresponds to the tape drive number. System Administration allows you to work with tape drives with rewind; no rewind is not supported.

SCSI-based tapes support the convention for naming and minor numbers in the format: */dev/c0[tx]d0m[n]* or */dev/c0[tx]d0h[n]*. The fields are described in Figure 12-4:

c0	controller number
[tx]	optional target controller
d0	tape drive number
m or h	density
[n]	no rewind

Figure 12–4 SCSI Tape Drive Device Name

Disk Subsystem

Each disk has two listings in */dev*: one as a block device and one as a character (raw) device. The special files for block disk devices are in the */dev/dsk* directory; the special files for raw disk devices are in the */dev/rdsk* directory. Each disk partition has a separate special file. Each disk drive also has entries in */dev/SA* and */dev/rSA* for block and character devices, respectively. These are used by System Administration to access disks, and are created with the **sysadm**(1) **mkdevdsk** command. The *SA rSA* device nodes are different for SCSI disks.

The common method for identifying disk device files has been interpreted for SCSI disks is similar to that of the SCSI tape drive. The format is */dev/c0[tx]d0s0* and is described in Figure 12-5.

c0	controller number
[tx]	target controller
d0	disk drive number
s0	section number

Figure 12–5 Disk Drive Device Name

The traditional name of a disk special file is the disk number and the partition number separated by an "s". For example, the special file for disk 1, partition 0 is *1s0*. If a disk drive has 8 physical partitions, they are numbered (named) 0 through 7 on each drive. The first disk drive in the system is number 0.

The minor number of a disk special file also identifies the disk and partition number with which the file is associated. Frequently, however, the minor numbers are assigned for the disk controller (which may control several disks) rather than the individual disks. For each controller, minor numbers start at 0 and increment by 1 to correspond to the partitions on the disks. The first disk on the controller has minor numbers 0 through 7, the second disk on the controller has minor numbers 8 through 15. So, partitions 0 through 7 on Disk 0 on CONTROLLER 0 have minor numbers 0 through 7, and partitions 0 through 7 on Disk 1 have minor numbers 8 through 15. If you had a second controller, the first disk on that controller would have minor numbers 0 through 7, but the major number would be different than for disks under controller 0.

The corresponding files for the raw disks have the same names and major and minor numbers but are located in the */dev/rdsk* directory.

The */dev/SA* and */dev/rSA* directories also have regular ASCII files for fixed disk devices, named *hddisk#*, where # corresponds to the disk drive number. These contain an ASCII character string which defines the type of disk this is. Because these are regular files, not special files, they do not have major and minor numbers.

After the */dev/dsk* and */dev/rdsk* files are created, use the **sysadm**(1) **mkdevdsk** or the **mknod**(1M) command to create the *rmdisk#* and *hddisk#* files in the */dev/SA* and */dev/rSA* directories. Figure 12-6 describes how minor numbers are formed on the 3B15 and 3B4000 computers.

Type	Special File	Minor Number =
Block access	*/dev/dsk/?x?*	Partition per controller
Character/raw access	*/dev/rdsk/?x?*	Partition per controller
Block access (**sysadm**)	*/dev/SA/hddisk?*	none
Character/raw access (**sysadm**)	*/dev/rSA/hddisk?*	none

Figure 12−6 3B15 or 3B4000 MP Minor Numbers and Names for Disk Devices

Other Devices

Minor numbers for other devices are assigned in a number of different ways. Several of the drivers that are released with UNIX System V (such as *errlog*, *swap*, and *dump*) have major and minor numbers that correspond to the disk partition they use; for instance, the major and minor numbers of */dev/swap* are the same as the major and minor numbers of the disk partition used as the swap device.

In some cases, the minor number of a software driver has little meaning and can be assigned any value.

Access Permissions for Special Device Files

The special device files used for drivers have access permissions, owners, and groups like any other file. Assigning appropriate values to these fields is critical for maintaining system security.

You must have super-user permissions to create special files with the **mknod** command. You can change the group with the **chgrp(1M)** command, and change the owner of a file with the **chown(1M)** command. The format for these two commands is:

chgrp *new-group special-file-name*
chown *new-owner special-file-name*

The default permissions are those specified by **umask** (in the */etc/system* file or in the *root .profile* file), usually 644. Permission modes can be modified with the **chmod**(1) command. Default permission modes can be modified with the **umask**(1) command.

chmod *new-mode special-file-name*

Figure 12-7 summarizes the recommended permissions, owner, and group for standard types of devices. The following sections discuss this in more detail.

Subsystem	Device	Mode	Owner	Group
Terminal	terminal (idle)	622 or 600	root	sys
	printer	200	lp	sys
	networks	644	uucp[4]	any
Disk	*/dev/rdsk* directory */dev/dsk* directory	755 755	root root	sys sys
	disk files	400	root	sys
Tape	*/dev/rmt* directory */dev/mt* directory	755 755	root root	sys sys
	tape files	666 or 600	root	sys

Figure 12–7 Typical Access Permissions for Special Device Files

Terminal Subsystem — Terminals

When a user logs on to the terminal port, that user becomes the owner and group for the port. The mode is 600 if the terminal is not open for writing from other users (**mesg n**) or 622 if it is. An active terminal should not normally be open for reading by other users, since this would enable other users to capture everything typed at or printed on the terminal. If wider permissions are necessary, any user can modify the mode of the terminal port to which s/he is logged in.

Some terminal special files retain the last user as the group when the user logs off, others will revert to the *sys* group. In any event, the idle terminal always reverts to an owner of root and mode 666.

Terminal Subsystem — Networks

Access permissions for networks should be considered very carefully, since system security is most easily compromised through network connections. The network itself is the owner. For instance, ACU nodes are usually owned by *uucp*. If another networking application needs to use the ACU, the software could execute a **setuid(uucp)**. The group can be left as the default *sys* or changed to match the owner.

The mode of networking devices must be determined according to how applications will access the network. If the networking connection is only for administrative programs, you can assign the secure mode of 600. If, however, application programs that

4. an example network name. This also could be the name for other networks.

understand the protocol will be accessing the network, you may require a 666 mode. If only a few users need to access the network, you can use the *group* modes. Most networks have a background program that writes to and reads from the special device file. Users rarely access it directly.

Terminal Subsystem — Printers

Special device files for printers are owned by **lp**; the group can be changed to **lp** or left as the default **sys**. Normally print jobs will run only through the **lpspooler**, so the 600 mode is adequate. If you have applications that will bypass **lpspooler** to go to the printer, you may need to set the mode to 644. However, read permissions are not necessary on a printer so you can set it to 200.

Disk Subsystem

The mode of a special device file for a disk only controls access permission to the physical disk. Once the disk is mounted, access to that disk is controlled by the file subsystem and the access permissions of each individual file. Special device files for disks have 400 permission, allowing reading and writing of the raw disk only by the owner (*root*). If read/write privileges were granted to others, the UNIX system security of all files on that disk would be subverted, since any user could read and write the contents of the disk without going through the file system. Application program may require different permission modes and ownership.

Tape Subsystem

Access permissions for tapes can vary from site to site. The most secure option is to use 600 permission, which will enable the superuser to use the tape but no one else to access it. The least secure option is to use 666 permission, which allows all users to read and write directly to/from that drive. Realize, however, that 666 permission will enable any user to read the information on that drive directly; for instance, when a tape is mounted for backup, a user could read all the information off that tape, thus accessing files that might contain sensitive information.

If several users need to access a tape drive, you could make those users part of the *sys* group or set up a group of users who need to access the tape and make that the group for the drive. By giving that drive 660 permission, these users would be able to access the tape without opening up access to the world.

The **sysadm mkdevdsk** or **mknod** command creates entries in the */dev/SA* and */dev/rSA* directories for removable disks and tape. The corresponding */dev* entries must be created first, either through **sysadm mkdevdsk** and **sysadm mkdevmt** or with the **mknod** command; the */dev/SA /dev/rSA* entries are then linked to the appropriate */dev* special files. In order to use the System Administration commands for disks and tapes, you must have this directory.

Adding to a Prototype File

On a 3B4000 adjunct processor a device file is created in three ways:

- With **mknod**(1M) in the *adj/pe#/dev* directory on the adjunct processor
- With **mknod** in the */dev* directory on the 3B4000 Master Processor
- By adding an entry to the */adj/pe#/prototype* file.

Use the directions for **mknod** discussed earlier in the "Creating a Device File" section to create special device files for the first two items. The third item is discussed in this section.

Each adjunct processor has a prototype file (*/adj/pe#/prototype*) used to configure the incore file system at boot time. This file specifies the size and contents of the incore file system. The prototype file is only activated after the adjunct processor is rebooted.

The prototype file contains a single line for each device for the adjunct processor. A prototype file line is in this format:

device-name type bits modes owner-ID group-owner-ID major-num minor-num

For example

device-name	*type*	*bits*	*modes*	*owner-ID*	*group-owner-ID*	*major-num*	*minor-num*
icfs	b	—	640	0	0	66	0

Figure 12-8 lists an excerpt from a sample prototype file.

```
icfs b--640 0 0 66 0
mem c--440 0 0 19 0
kmem c--440 0 0 19 1
null c--666 0 0 19 2
error c--660 0 0 16 0
dsk d--755 0 0
           c0t1d0s0 b--400 0 0 113 0
           c0t1d0s1 b--400 0 0 113 1
           c0t1d0s6 b--400 0 0 113 6
           c0t1d0s7 b--400 0 0 113 7
           c0t1d0s8 b--400 0 0 113 8
           c0t2d0s0 b--400 0 0 114 0
           c0t2d0s1 b--400 0 0 114 1
           (Additional Entries)
           $
```

Figure 12–8 Excerpt from Sample Prototype File

Adding Information to the /etc/system File

When you are installing a software driver for the first time, you must insert a line in the */etc/system* file so that the driver is included when the new version of UNIX is created system. This step is not required for a hardware driver.

The */etc/system* file is used to initially configure or to reconfigure the UNIX operating system. After the system configures, an operating system image is made in memory and booted. Then, by invoking the **/etc/mkunix** program (done automatically on the SBC and 3B2 computers), a bootable image of the operating system is created which, by convention, is named */unix*. The */unix* file can then be used to boot the system quickly.

Among other kinds of information, the */etc/system* file lists the drivers that are to be included when the system is configured. In order to configure your driver into the system, you must include the name of your driver in the */etc/system* file and then reboot the system from this file.

Edit the system file (*/etc/system*) and add an INCLUDE line for your driver to the end of the file. Comments can be added by placing an asterisk (*) in the first column. The new lines in an example system file are

```
*
* Include line for mydriver.  Added 1/25/88 by Jane Doe.
*
INCLUDE:MYDRIVER
```

The sections of */etc/system* are referred to as lines, even though many of them have several lines. The **system**(4) manual page explains all the lines that are in */etc/system*. Discussed here are only those lines used for drivers. They are

EXCLUDE Specifies hardware listed in the EDT that should not be configured. This line can list hardware for which the software driver is not working or a board that needs repair and is affecting system stability.

INCLUDE Lists drivers with files in the */boot* directory but no corresponding device in the EDT, typically software drivers.

Creating Diagnostics Files

On the 3B2 computer, the SBC, or the 3B4000 ACP, if you are installing a new circuit board (feature card), obtain the diagnostics files from your diagnostics developer or create the files yourself. Refer to Appendix B for information on how to write or modify diagnostics files and to Section D8X of the *BCI Driver Reference Manual*. If the diagnostics files are not available or if you would prefer to install your driver before the files are available, execute the following commands:

cd /dgn
ln SBD *name*
ln X.SBD *X.name*

Linking to the system board (SBD) diagnostics files has no effect on the system; when your circuit board is tested, the system board is tested instead. This solution should only be regarded as temporary; no product is well-served by deluding the operating system.

If you are installing an existing circuit board, ensure that there are two files in the */dgn* directory for your driver. The first diagnostic file (required in the */dgn* directory) has the same name as the master file for your driver, except that the diagnostics file name is in all upper case. The second required diagnostics file has the same name as the first, except that the second file is preceded with "*X.*".

Adding a Device to the EDT

On the 3B2 computer, the SBC, or the 3B4000 MP equipped with SCSI, use the **edittbl**(1M) command to update the */dgn/edt_data* table to reflect the new device.

NOTE: Two **edittbl**(1M) exist, one for non-SCSI and the other for editing the SCSI Equipped Device Table (EDT). Use the command appropriate for your system. **edittbl** is in the */dgn* directory (for non-SCSI editing) and in the */etc/scsi.d* directory for SCSI. Refer to Appendix A for information about using **edittbl**.

Preparing Pump Files

On the 3B2 computer or the SBC, if intelligent boards need to be pumped with operational code, copy the pump code file to */lib/pump/<board-name>* and write a shell script to execute the pump code file. Place the shell script in the */etc/rc2.d* directory. (Examine the shell scripts in the */etc/rc2.d* directory for information on creating a shell script for your pump code.) The shell script is executed at boot time. The permission modes should be 500 with both owner and group being *root*.

On the 3B4000 MP, copy the pump code file to the */lib/bootpump.d* directory.

Installing an Existing Driver

This section describes how to install both hardware and software drivers on these computers

- Single Board Computer (SBC) and the 3B2 computers
- 3B15 computer or the 3B4000 Master Processor
- 3B4000 adjunct processor
- 3B15 computer

Separate installation instructions are provided for each computer by the type of driver being installed. Preceding these sections is information about how to compile a driver program for installation. This step is common to all computer types and is repeated many times in the process of installing a driver.

Before starting the driver installation, you should be familiar with the material in the last section. This section assumes that you have moved the driver code to a source directory, created a master file, and created any device files that are needed. If you are installing a driver for the first time and have not completed these activities, return to the last section, "Installing a Driver for the First Time", and ensure that all pre-installation files are in place.

CAUTION: Before installing a driver, you must back up */unix*. Failure to do so can mean performing a complete install of your original pristine software and rerunning all add-on installations. This process could require many hours of system down-time to complete. Select any name for the copy and write the name down. Should the need arise that you need to boot from the alternative file name, you will not have access to the disk to determine the file's name. Use the **mv**(1) command to move */unix* to another name and then use **cp**(1) to copy the file back to */unix*. This ensures that when the system is booted, a new version of the operating system is generated. An example set of commands for this procedure are

```
# cd /
# mv /unix /old.unix
# cp /old.unix /unix
#
```

In addition, the following files should be copied before starting a driver installation:

File or Directory	Description
/boot directory	bootable object files used for building a new version of the operating system
/etc/master.d directory	system configuration information
/etc/system file	indicates which files to include in a new version of the operating system

Figure 12–9 Files to Copy Before Installing a Driver

The files in the */boot* directory, those in the */etc/master.d* directory, and the */etc/system* file are backed up for safe keeping and are seldom ever in jeopardy. However, if these files were erased, restoring them could take many hours of loading the original system software and then rerunning all add-on installations. The minutes of copying these files now can save you hours or days of time later on.

Compiling a Driver for Installation

You can use the normal **cc**(1) command to ensure that your driver is free of syntax errors. However, for driver installation, more **cc** options are used to ensure that the driver produces the correct output and that the output files are in a format compatible with debugging tools. The compile line is

cc -c -DINKERNEL -D*computer* `-O` *file.c*

The options are

-c suppress the link editing phase of the compilation and do not remove any produced object files

-DINKERNEL enable access to macros and parts of source code enclosed as follows

```
#ifdef INKERNEL
. . .
#endif
```

-D*computer* substitute your computer type for *computer*. Figure 12-10 lists the available choices.

Name	Computer
ADJUNCT	Any type of adjunct processor
u3b15	3B15 or 3B4000 MP
u3b2	3B2 300, 400, 500, 600, and SBC
u3bacp	3B4000 ACP adjunct
u3badp	3B4000 ADP adjunct
u3beadp	3B4000 EADP adjunct

Figure 12−10 Computer Types

Use ADJUNCT for all types of adjunct processors; use **u3bacp, u3badp,** or **u3beadp** for the specific adjunct processor type.

This option enables access to macros and source code enclosed, for example, for a 3B2 computer as follows

```
#if u3b2
. . .
endif
```

-O optimize the code. (Do not use on SBC drivers.)

Other Options

Other options that you may need are

-r when compiling more than one *.c* file together to create a single driver object file.

-I when you need to specify the location of the header files when the location differs from */usr/include/sys*.

-Dm32b if the driver may have code ported from a 16-bit computer to a 3B15 computer or 3B4000 computer and the code is enclosed in this unit

```
#ifdef  m32b
. . .
endif
```

NOTE: When debugging is complete, use **strip**(1) to strip symbol and line number information from the resulting *.o* file. This saves space in the resulting bootable image.

Installing an SBC or 3B2 Computer Hardware Driver

Figure 12-11 provides a checklist for installing a hardware driver. Included in the checklist are steps from the previous section on installing a driver for the first time. Photocopy this page and include it with the documentation packet for your driver.

Step#	Description	Perform	Completed?
1	create a master file	once	
2	create necessary device files	once	
3	create diagnostics files	once	
4	update the */dgn/edt_data* file	once	
5	put pump code files in special directory	once	
6	back up */unix* before each installation	as needed	
7	compile driver source code	as needed	
8	create a bootable object file	as needed	
9	run **touch**(1) on */etc/system*	as needed	
10	run **shutdown**(1M)	as needed	

Figure 12-11 SBC or 3B2 Computer Hardware Driver Installation Checklist

The **Perform** column indicates how many times you should perform a step in preparation for installing the driver. Steps performed once are found in the previous section, "Installing a Driver for the First Time"; steps that are performed as needed are explained in this section. (Compiling a driver is explained in the previous section.)

Install an SBC or a 3B2 computer hardware driver as follows:

Step 8 Create a bootable object file for your driver with the **mkboot** command.

The command syntax for **mkboot**(1M) is

/etc/mkboot *file-name*.**o**

This command creates the */boot/file-name* file. Refer to the **mkboot**(1M) manual page for more information on command options.

Step 9 Run the **touch**(1) command on */etc/system*. This command sets the date of last modification to the current date.

Step 10 Bring the system down with the **shutdown**(1M) command (from the root directory)

shutdown -g0 -y -i6

If the installation is successful, no error messages are displayed and the "Console Login:" prompt is displayed. If the installation fails, turn to Chapter 13 to debug your driver. To recover your system for debugging, shut down your computer as follows:

shutdown -g0 -y -i5

At the FIRMWARE MODE prompt, enter the Maintenance and Control Program (MCP) password, usually **mcp** and press the RETURN key. At the following prompt, enter **/old.unix** (assuming that you backed up the previous version of */unix* as explained at the start of this section).

```
Enter name of program to execute [   ]:
```

Installing an SBC or 3B2 Computer Software Driver

Figure 12-12 provides a checklist for installing a software driver. Included in the checklist are steps from the previous section on installing a driver for the first time. Photocopy this page and include with the documentation packet for your driver.

Step#	Description	Perform	Completed?
1	create a master file	once	
2	create necessary device files	once	
3	insert an INCLUDE line in */etc/system*	once	
4	backup */unix* before each installation	as needed	
5	compile driver source code	as needed	
6	create a bootable object file	as needed	
7	run **touch**(1) on */etc/system*	as needed	
8	run **shutdown**(1M)	as needed	

Figure 12-12 SBC or 3B2 Computer Software Driver Installation Checklist

The **Perform** column indicates how many times the step should be performed. Steps performed once are found in the previous section, "Installing a Driver for the First Time"; steps that are performed as needed are explained in this section. (Compiling a driver is explained at the start of this section.)

Install an SBC or a 3B2 computer software driver as follows:

Step 6 Create a bootable object file with the **drvinstall**(1M) command. (Once a major device number is assigned, you can use either **drvinstall** or **mkboot** as shown in the sections on installing a hardware driver.) The **drvinstall** command has the following format:

/etc/drvinstall -d *pathname-of-object-file* **-v1.0**

Use the **-d** option to identify the pathname of the input object file. Use the **-v1.0** argument (required) to specify the version number of **drvinstall**. When run, **drvinstall** returns the major number. **drvinstall** creates a new major number if a dash (-) is encoded in the SOFT column of the master file. If a number is already in the SOFT field, **drvinstall** echoes that number as the return value. If a major number is created, **drvinstall** replaces the dash under SOFT in the master file with the new major number. **drvinstall** creates a bootable driver file in the */boot* directory in the form of the driver name in upper case.

NOTE: **drvinstall** can be run from any directory. However, **drvinstall** does not accept a dot (.) as the directory name. It only accepts the full pathname of the input object file created with the appropriate **cc**(1) command. An input object file compiled by **cc** must never be placed in the *boot* directory. Therefore, put the input object file elsewhere and always use **drvinstall** with the **-d** option.

If key files that **drvinstall** accesses are located in non-standard locations or are for adjunct processors, identify the files to **drvinstall** with the following options:

file	*default*	**option**
master file	*/etc/master.d*	**-m**
system file	*/etc/system*	**-s**
output directory	*/boot*	**-o**

Step 7 If you are installing a previously installed driver, run the **touch**(1) command on */etc/system*. If this is the first installation of a driver, skip this step. When you added the INCLUDE line to */etc/system*, you achieved the same purpose as this step. This command sets the date of last modification to the current date.

Step 8 Bring the system down with the **shutdown**(1M) command (from the root directory)

```
shutdown -g0 -y -i6
```

If the installation is successful, no error messages are displayed and the "Console Login:" prompt is displayed. If the installation fails, turn to Chapter 13 to debug your driver. To recover your system for debugging, shut down your computer as follows:

```
shutdown -g0 -y -i5
```

At the FIRMWARE MODE prompt, enter the Maintenance and Control Program (MCP) password, usually **mcp** and press the RETURN key. At the following prompt, enter **/old.unix** (assuming that you backed up the previous version of */unix* as explained at the start of this section).

```
Enter name of program to execute [  ]:
```

Installing a 3B15 Computer or 3B4000 MP Hardware Driver

Figure 12-13 provides a checklist for installing a hardware driver. Included in the checklist are steps from the previous section on installing a driver for the first time. Photocopy this page and include it with the documentation packet for your driver.

Step#	Description	Perform	Completed?
1	create a master file	once	
2	create necessary device files	once	
4	SCSI only: update the */dgn/edt_data* file	once	
5	3B4000 MP only: put pump code files in */lib/bootpump.d*	once	
6	back up */unix* before each installation	as needed	
7	compile driver source code	as needed	
8	create a bootable object file	as needed	
9	run **touch**(1) on */etc/system*	as needed	
10	run **shutdown**(1M)	as needed	
11	run **mkunix**(1M)	as needed	

Figure 12−13 3B15 Computer or 3B4000 MP Hardware Driver Installation Checklist

The **Perform** column indicates how many times you should perform a step in preparation for installing the driver. Steps performed once are found in the previous section, "Installing a Driver for the First Time"; steps that are performed as needed are explained in this section. (Compiling a driver is explained in the previous section.)

Install a 3B15 computer or 3B4000 MP hardware driver as follows:

Step 8 Create a bootable object file for your driver with the **mkboot** command.

The command syntax for **mkboot**(1M) is

/etc/mkboot *file-name*.**o**

This command creates the */boot/file-name* file. Refer to the **mkboot**(1M) manual page for more information on command options.

Step 9 Run the **touch**(1) command on */etc/system*. This command sets the date of last modification to the current date.

Step 10 Bring the system down with the **shutdown**(1M) command (from the root directory)

```
shutdown -g0 -y -i6
```

If the installation is successful, no error messages are displayed and the system boots normally. If the installation fails, turn to Chapter 13 to debug your driver. To recover your system for debugging, shut down your computer as follows:

```
shutdown -g0 -y -i5
```

At the following prompt, enter **/old.unix** (assuming that you backed up the previous version of */unix* as explained at the start of this section).

```
Enter path name:
```

Step 11 After your driver is working and you want to preserve your driver in the */unix* file, run **mkunix** to create a new version of the operating system. This step must be performed each time the driver is installed if you are going to bring the computer down to test firmware.

Installing a 3B15 Computer or 3B4000 MP Software Driver

Figure 12-14 provides a checklist for installing a software driver. Included in the checklist are steps from the previous section on installing a driver for the first time. Photocopy this page and include with the documentation packet for your driver.

Step#	Description	Perform	Completed?
1	create a master file	once	
2	create necessary device files	once	
3	insert an INCLUDE line in */etc/system*	once	
4	backup */unix* before each installation	as needed	
5	compile driver source code	as needed	
6	create a bootable object file	as needed	
7	run **touch**(1) on */etc/system*	as needed	
8	run **shutdown**(1M)	as needed	
9	run **mkunix**(1M)	as needed	

Figure 12−14 3B15 Computer or 3B4000 MP Software Driver Installation Checklist

The **Perform** column indicates how many times the step should be performed. Steps performed once are found in the previous section, "Installing a Driver for the First Time"; steps that are performed as needed are explained in this section. (Compiling a driver is explained at the start of this section.)

Install a 3B15 computer or 3B4000 MP software driver as follows:

Step 6 Create a bootable object file with the **drvinstall**(1M) command. (Once a major device number is assigned, you can use either **drvinstall** or **mkboot** as shown in the sections on installing a hardware driver.) The **drvinstall** command has the following format:

/etc/drvinstall -d *pathname-of-object-file* **-v1.0**

Use the **-d** option to identify the pathname of the input object file. Use the **-v1.0** argument (required) to specify the version number of **drvinstall**. When run, **drvinstall** returns the major number. **drvinstall** creates a new major number if a dash (-) is encoded in the SOFT column of the master file. If a number is already in the SOFT field, **drvinstall** echoes that number as the return value. If a major number is created, **drvinstall** replaces the dash under SOFT in the master file with the new major number. **drvinstall** creates a bootable driver file in the */boot* directory in the form of the driver name in upper case.

NOTE: **drvinstall** can be run from any directory. However, **drvinstall** does not accept a dot (.) as the directory name. It only accepts the full pathname of the input object file created with the appropriate **cc**(1) command. An input object file compiled by **cc** must never be placed in the *boot* directory. Therefore, put the input object file elsewhere and always use **drvinstall** with the **-d** option.

If key files that **drvinstall** accesses are located in non-standard locations or are for adjunct processors, identify the files to **drvinstall** with the following options:

file	*default*	**option**
master file	*/etc/master.d*	**-m**
system file	*/etc/system*	**-s**
output directory	*/boot*	**-o**

Step 7 If you are installing a previously installed driver, run the **touch**(1) command on */etc/system*. If this is the first installation of a driver, skip this step. When you added the INCLUDE line to */etc/system*, you achieved the same purpose as this step. This command sets the date of last modification to the current date.

Step 8 Bring the system down with the **shutdown**(1M) command (from the root directory)

```
shutdown -g0 -y -i6
```

If the installation is successful, no error messages are displayed and the system boots normally. If the installation fails, turn to Chapter 13 to debug your driver. To recover your system for debugging, shut down your computer as follows:

```
shutdown -g0 -y -i5
```

At the following prompt, enter **/old.unix** (assuming that you backed up the previous version of */unix* as explained at the start of this section).

```
Enter path name:
```

Step 9 After your driver is working and you want to preserve your driver in the */unix* file, run **mkunix** to create a new version of the operating system.

Installing a 3B4000 Adjunct Processor Hardware Driver

Figure 12-15 provides a checklist for installing a hardware driver. Included in the checklist are steps from the previous section on installing a driver for the first time. Photocopy this page and include it with the documentation packet for your driver.

Step#	Description	Perform	Completed?
1	create adjunct master file	once	
2	create device files on the Master Processor	once	
3	update adjunct prototype file	once	
4	create adjunct diagnostics files	once	
5	update adjunct *edt_data* file	once	
6	put pump code files in special directory	once	
7	compile driver source code	as needed	
8	create adjunct bootable object file	as needed	
9	run **touch**(1) on */adj/pe#/etc/system*	as needed	
10	stop adjunct processor	as needed	
11	restart adjunct processor	as needed	

Figure 12−15 3B4000 Adjunct Processor Hardware Driver Installation Checklist

The **Perform** column indicates how many times you should perform a step in preparation for installing the driver. Steps performed once are found in the previous section, "Installing a Driver for the First Time"; steps that are performed as needed are explained in this section. (Compiling a driver is explained in the previous section.)

Install an adjunct processor hardware driver as follows:

Step 8 Create a bootable object file for your driver with the **mkboot** command.

The command syntax for **mkboot**(1M) is

/etc/mkboot -P *pe# file-name*.**o**

This command creates the */adj/pe#/boot/file-name* file. Refer to the **mkboot**(1M) manual page for more information on command options.

Step 9 Run the **touch**(1) command on */adj/pe#/etc/system*. This command sets the date of last modification to the current date.

Step 10 Take the adjunct processor out-of-service with the **stopape**(1M) command. For example, to stop adjunct processing element #120, enter

```
# /etc/stopape -P 120
```

If a file system on the adjunct has active processes, the adjunct is not stopped unless you add the **-K** option to the command.

Step 11 Restore an out-of-service adjunct processor with the **bootape**(1M) command. **bootape** creates a new version of the operating system if the date on the */adj/pe#/etc/system* file has been updated with **touch** or */adj/pe#/unix* file has been moved and copied back. For example, to boot adjunct processing element 120, enter

```
# bootape -P 120
```

Installing a 3B4000 Adjunct Processor Software Driver

Figure 12-16 provides a checklist for installing a software driver. Included in the checklist are steps from the previous section on installing a driver for the first time. Photocopy this page and include with the documentation packet for your driver.

Step#	Description	Perform	Completed?
1	create adjunct master file	once	
2	create device files on the Master Processor	once	
3	insert INCLUDE line in adjunct system file	once	
4	update adjunct prototype file	once	
5	compile driver source code	as needed	
6	create bootable object file	as needed	
7	run **touch**(1) on system file	as needed	
8	stop adjunct processor	as needed	
9	restart adjunct processor	as needed	

Figure 12-16 3B4000 Adjunct Processor Software Driver Installation Checklist

The **Perform** column indicates how many times the step should be performed. Steps performed once are found in the previous section, "Installing a Driver for the First Time"; steps that are performed as needed are explained in this section. (Compiling a driver is explained at the start of this section.)

Install adjunct processor software driver as follows:

Step 6 Create a bootable object file with the **drvinstall**(1M) command. (Once a major device number is assigned, you can use either **drvinstall** or **mkboot** as shown in the sections on installing a hardware driver.) The **drvinstall** command has the following format:

/etc/drvinstall -P *pe#* **-d** *pathname-of-object-file* **-v1.0**

Use the **-d** option to identify the pathname of the input object file. Use the **-v1.0** argument (required) to specify the version number of **drvinstall**. When run, **drvinstall** returns the major number. **drvinstall** creates a new major number if a dash (-) is encoded in the SOFT column of the master file. If a number is already in the SOFT field, **drvinstall** echoes that number as the return value. If a major number is created, **drvinstall** replaces the dash under SOFT in the master file with the new major number. **drvinstall** creates a bootable driver file in the */boot* directory in the form of the driver name in upper case.

NOTE: **drvinstall** can be run from any directory. However, **drvinstall** does not accept a dot (.) as the directory name. It only accepts the full pathname of the input object file created with the appropriate **cc**(1) command. An input object file compiled by **cc** must never be placed in the *boot* directory. Therefore, put the input object file elsewhere and always use **drvinstall** with the **-d** option.

If key files that **drvinstall** accesses are located in non-standard locations or are for adjunct processors, identify the files to **drvinstall** with the following options:

file	*default*	**option**
master file	*/etc/master.d*	**-m**
system file	*/etc/system*	**-s**
output directory	*/boot*	**-o**

Step 7 If you are installing a previously installed driver, run the **touch**(1) command on */etc/system*. If this is the first installation of a driver, skip this step. When you added the INCLUDE line to */etc/system*, you achieved the same purpose as this step. This command sets the date of last modification to the current date.

Step 8 Take the adjunct processor out-of-service with the **stopape**(1M) command. For example, to stop adjunct processing element #120, enter

```
# /etc/stopape -P 120
```

If a file system on the adjunct has active processes, the adjunct is not stopped unless you add the **-K** option to the command.

Step 9 Restore an out-of-service adjunct processor with the **bootape**(1M) command. **bootape** creates a new version of the operating system if the date on the */adj/pe#/etc/system* file has been updated with **touch** or */adj/pe#/unix* file has been moved and copied back. For example, to boot adjunct processing element 120, enter

```
# bootape -P 120
```

Installing a Driver for Testing

During the testing and debugging phase, you may want to install your driver in an "unofficial" manner so you can easily restore the system to a normal operating state, without the driver. How you install your driver during this phase will be determined by considerations such as whether the system is dedicated to development or also a production machine and whether other people are developing other drivers on this same machine.

On the 3B15 or 3B4000 computers, you can bring your computer up in "magic mode". At the "Enter pathname" prompt, enter

magic mode *boot-dir*

Where *boot-dir* is an alternative */boot* directory. You are then prompted for the system file name. The configuration is generated and then a load map is listed. Control returns at firmware mode. This is useful when using specialized debugging tools that permit break point setting and memory examination. If your site supports such a tool or if you wish to configure a system with an alternative */boot* directory, you may wish to substitute this procedure in the following installation steps when booting the computer is necessary.

This section recommends installation steps to take if you are developing your driver on a computer that is used for other purposes, that will need to be restored to normal operation in between your testing times

1. If it is necessary to modify the */etc/system* file for your driver, make a copy of it (such as */etc/janesystem*). The installation will be performed on the */etc/system* file. Should something go wrong, copy */etc/janesystem* back to */etc/system* to restore the system file to its previous state.

2. For hardware drivers, add an EXCLUDE line to the */etc/system* file. This will prevent your driver from being configured when you boot from */etc/system*.

3. Copy the */unix* file to another name that is not currently in use (such as */holdunix*) or back it up to tape or floppy disk. Be sure you do not overwrite a copy of */unix* that someone else is holding.

4. If necessary, modify the */etc/inittab* or */etc/rc0* files or add scripts to the */etc/brc.d* or */etc/rc.d* directories. If you have to restore the system to normal operating status after your testing, you will need to remove these entries and files.

5. For software drivers, run the **/etc/drvinstall**(1M) command.

6. Create the special device files with the **mknod**(1M) command.

7 Create the master file in the */etc/master.d* directory, under a name such as *newmaster*. As an alternative, you can create a separate master directory and indicate it with the **mkboot -m** option. When installing one or a few drivers, using */etc/master.d* should not cause any problems. However, if you create an alternative master file directory, when you use **drvinstall**, specify the **-m** option so that the new master file directory is checked. In addition, if you are installing a software driver, you should be aware that since **drvinstall** selects the major number, you may have a duplicated major number. This may necessitate re-installation of your driver when you want to place your master file in */etc/master.d*.

Several installation tasks can be done once and used throughout the testing/debugging phase, while other tasks must be redone every time you modify the driver code.

1 Create the driver object code by compiling the driver source code. This should not be done in the */boot* directory, but in the development directory.

2 Run **mkboot** to create a bootable object file in */boot*. Run this from the development directory where you have created the *master.d* and driver object file. A sample command line is

/etc/mkboot *driver.o*

Installing a Driver in a Cross Environment

You can develop a driver for a different type of computer or UNIX System Release than the one on which you are developing; this is referred to as working in a *cross environment* or *native environment*. This discussion is restricted to UNIX System V Release 3.0 and later on the 3B2, 3B15, 3B4000, and SBC computers, although many of the principles can be applied to other situations.

For this discussion, *development machine* refers to the machine on which you are working; *target machine* refers to the other computer or operating system on which you want the driver to run.

To compile in a cross environment, you must have the following installed on your development machine:

- The C compiler and assembler for the target computer (cross Software Generation System — SGS)
- Set of system headers for the target computer

Installation of A Completed Driver

This section discusses the steps to take to officially install a driver on your own machine. If you intend to install this driver on a number of machines, you may want to follow the procedures in the section on *Packaging Installation and Removal Procedures* in Chapter 16.

Code Clean Up

Before officially installing the driver, you should clean up the code. You can remove statements used for debugging or surround the code in the conditional compile **#if ... #endif** statements. For example

```
#if DEBUG
        cmn_err(CE_CONT,"Starting Shutdown.0);
#endif
```

Specific items to look for in driver code include

- Remove or surround in **#if ... #endif** all **cmn_err** statements put in for tracing and debugging.
- Check that the text of **cmn_err** statements are clear and contain no spelling or grammatical errors.
- Remove or surround in **#if ... #endif** all calls to the TRACE driver.
- Check that the **sleep** priorities have been reset to an appropriate level for a production driver.
- Disable private logging and debugging utilities built into the driver.

In addition, you should check for the following items before releasing any software for production work

- Be sure that code is thoroughly commented.
- If appropriate, be sure that all unnecessary references to proprietary information and development names are removed from the comments.
- Check that the **#ident** statement is present and contains the appropriate version information. The information enclosed in the **#ident** statement is placed into the *.comment* section of an *a.out* file. This capability, known as an S-list, is useful for keeping software version information. Refer to the documentation that accompanied your "C" programming language utilities for more information.
- If you are copyrighting the software, this may be the time to change all copyright notices to reflect a final product rather than work in progress. Check with your own legal counsel about when to take this step.

Removing a Driver

To remove a driver from the system, you must remove (or restore to their former state) all files that you modified to add the driver to the system. The procedure is

1. For hardware drivers, physically remove the hardware device and associated subdevices from the system.

2. For hardware devices on the SBC, 3B2 computers, and the 3B4000 ACP, edit the *edt_data* file to remove the device and its associated subdevices. If necessary, remove any associated diagnostics files from the */dgn* directory.

3. For hardware devices, delete the files in the */etc/master.d* and the */boot* directories for your driver.

4. For software drivers, run one of the following commands:

   /etc/drvinstall -u -d*object*
   /etc/drvinstall -u -m*master*

   This removes the bootable object file from the */boot* directory, replaces the major number in the appropriate */etc/master.d* file with a dash, thus unassigning the major number, and removes the INCLUDE line from the */etc/system* file.

5. Remove special device files and any */etc/rc** or */etc/brc.d* scripts you created. This will vary with the functionality. For instance, if the script will actually be looking for the kernel routines from the driver, it must be removed. Other drivers, such as those that remake special device files, may be harmless if not removed. All such files should be removed (or restored) when you permanently remove the driver from the system.

You can temporarily remove a driver from the system (such as during testing and debugging) by

Hardware Driver: Add an EXCLUDE line to the */etc/system* file for the hardware device.

Software Driver: Remove the INCLUDE line for the software device from the */etc/system* file.

After altering the system file, reboot your system and make a new */unix* file. The new */unix* should be identical to the */unix* you saved before adding the new drivers.

Chapter 13: Testing and Debugging the Driver

Contents

Introduction

Debugging a driver is largely a process of analyzing the code and thinking about what could have caused the problem. The UNIX operating system includes some tools that may help, but because the driver operates at the kernel level, the tools can only provide limited information. For this reason, it is useful to do simulation testing of the driver as a user-level process before installing it and beginning formal testing.

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

The six aspects of debugging a driver are

1. Test the basic functionality of the hardware (hardware drivers only).
2. Debug the C code with the standard C programming language debugging tools. (This is not discussed here.)
3. Simulation test the driver at the user level.
4. Install the driver and ensure that the system can be booted with the driver in place.
5. Test the functionality of the driver in single-user mode.
6. Test the driver on a fully-loaded system (integration testing).

During the first phases of testing, remember that your driver code is probably not perfect and that bugs in the driver code may well panic or damage the system, even parts of the system that may seem unrelated to your driver. Testing should be done when no other users are on the system and all production data files are backed up.

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

Testing the Hardware

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

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

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

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

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

Testing Driver Functionality

The process of testing driver functionality is piecemeal: you have to take small pieces of your driver and test them individually, building up to the implementation of your complete driver. The UNIX operating system provides tools, such as **crash**(1M) (which can be used either for a post mortem analysis or for interactive monitoring of the driver) and the **trace** driver (for the 3B4000 computer), to help you.

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

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

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

After you are comfortable that both the hardware and software behaves as it should during error situations, it is time to concentrate on formal performance testing. This is discussed in Chapter 14, "Performance Considerations."

Getting Started

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

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

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

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

Using the instructions in Chapter 12, "Installation," install your driver. If the UNIX system does not come up, divide your driver into separate sections and install these separately until you find the problem. Fix the problem and install the driver.

After the driver is installed, run **mkunix**(1M) to create a new */unix* file.

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

Using cmn_err

Use the **cmn_err**(D3X) function to put debugging comments in the driver code; when the driver executes, you can use these to tell what part of the driver is executing. The **cmn_err** function is similar to the **printf**(2) system call but it executes from inside the kernel. For instructions on using the **cmn_err** statement, see Chapter 11, "Error Reporting."

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

Calculations and **cmn_err** statements that are for debugging and other testing should be coded within conditional compiler statements in the driver. This saves you the task of removing extraneous code when you release the driver for production, and makes that debugging code readily available should you need to troubleshoot the driver after it is in the field.

You can provide separate code for different types of testing to which the driver will be subjected. For instance, you might use TEST for functionality testing, PERFON for minimal performance testing, and FULLPERF for full performance monitoring. Each of the testing options is then defined in the code as either 0 (turned off) or 1 (turned on), as illustrated in Figure 13-1.

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

Figure 13–1 Defining Test Options

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

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

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

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

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

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

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

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

Using crash to Debug a Driver

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

On the 3B4000 computer, the **crash** command is used with the **-P** *PE-number* option to specify an adjunct processing element. The **crash** command run without a **-P** option or with **-P 121** analyzes the Master Processor (MP) kernel. When running **crash** on an adjunct, the system uses the following files:

/adj/pe#/unix	for symbol table (located on MP)
/adj/pe#/dev/mem	for memory access (located on the adjunct)

Each invocation of **crash** can only look at one kernel. Should you need to view more than one kernel simultaneously, use a separate terminal or window to invoke **crash** on each kernel.

Saving the Core Image of Memory

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

The following table summarizes how to save the core image of memory on the various computers covered in this book:

Table 13–1 Saving Core Image of Memory

3omputer	*Command*	*Destination of Dump*
SBC	**sysdump(8)**	Floppy on 1st disk controller
3B2	**sysdump(8)**	Floppy disk(s) mounted on /dev/c0d0s6
3B15 and 3B4000 MP	**dump(8)**	Partition specified in */dev/dump*, as specified by DUMPDEV in */etc/system*, unless otherwise specified
3B4000 Adjunct	**adjdump(8)**	*adjdump.out* in current directory unless otherwise specified

On the SBC and 3B2 computer, you use a series of floppies to hold the memory dump. The system prompts you to load the next diskette. Be sure that these diskettes are labeled clearly so you can load them in the proper sequence when running **crash**. The label information should include the date and time of the dump.

On the 3B15 computer and 3B4000 Master Processor, the system automatically takes the dump when the automatic reboot feature is enabled. You should copy the contents of the */dev/dump* partition to a regular file after the system is rebooted to avoid overwriting the information. A common procedure is to create a directory, such as */usr/dumps*, to hold memory dumps. The regular files in this directory should have names that include date and time information and, for the 3B4000 computer, PE number.

On the 3B4000 Adjuncts, the MP must be running in either single- or multiuser state and the MAP must be running before you run **adjdump**. If necessary, start it with the **sysadm startmap** command.

For full instructions on running these commands, consult the administrative documentation for the appropriate system.

Before running **crash**, check that the memory dump is sane. Verify the following:

- The size of the dump file should match processor memory size.
- The **stat** function should give the correct system name, node, and release of the running operating system. Be sure that the UNIX system version agrees with the *namelist* file being used.
- The date and time of the crash reported from the dump file should be reasonable given the actual date and time of the system panic. Note that the dump may be usable even if this information is wrong.
- The PID, PPID, PGRP, UID, PRI, and CPU fields should have reasonable numbers when reported by the **proc** function. Note that the values will be decimal.
- The **user** function should not respond with a read error.

If these checks indicate that the memory dump is not sane, try to reproduce the error and take a new dump.

Initializing crash on the Memory Dump

To run **crash** on the core image of memory at the time the system panicked, you must have saved the core image before rebooting and the file containing the kernel bootable image (*/unix* file by default) that was running at the time of the crash. The **crash** command can be run by any user with read permission on the */dumpfile*.

The command to initialize **crash** is

/etc/crash -d *dumpfile* **[-n** *namelist***] [-w** *outputfile***]**

For a 3B4000 adjunct, use the **-P** *PE-number* option to specify an adjunct kernel. For example, to initialize **crash** on PE 8, the command is

/etc/crash -P 8 -d *dumpfile* **[-n** *namelist***] [-w** *outputfile***]**

When running a postmortem crash analysis, you must specify the file that contains the memory dump. On the SBC and 3B2 computer, you can run **crash** directly from the floppy disks by specifying **-d /dev/ifdsk06**, or you can first run **ldsysdump**(1M) to write the contents of the floppies to a file on hard disk and specify the name of that file.

If the bootable kernel image is named something other than */unix* (either because it was named something else at the time of the panic or because you copied it to another name after the panic), use

the **-n** option or the second positional parameter to specify that file name. If you want the output of **crash** to be written to a file rather than your terminal (standard output), use the **-w** option with the name of the file. Note that the output of a specific **crash** function can be redirected to a file even if you do not use the **-w** in the **crash** command line.

The first step in using **crash** to analyze a post mortem dump is to determine your program's offset. The technique for doing this is

1. Find the registers for your program, specifically the stack pointer.
2. Locate the stack and trace back through the stack to find the last routine called by your driver. The very last routine on the stack pointer should be the panic message that invoked the crash. Data in the stack previous to the crash can contain pointers to various parts of the kernel. You have to sift through the data in the stack to find the last routine called by your driver. This involves cross referencing between driver listings and the core dump using the **crash nm** function to examine the stack addresses until the information is found.
3. The offset is the difference between the program counter and where the last routine started.

From the program counter, you can determine from the name list the exact routine that was executing at the time of the failure. Going back to the disassembled listing of your driver, you can then determine the exact instruction that was running. You should then use the output of the **list** command to determine the exact line in your source file where the failure occurred.

In the postmortem dump, you will need the offset described previously. **crash** displays in absolute code segments without access to your program's symbolic constants. You must use your program's offset to determine where your program is in the kernel and to trace its flow.

Initializing crash on an Active System

Running **crash** on an active system is useful for checking the buffer pools, determining that the members of driver structures have correct values, and ensuring that all operations are synchronized. Interactive **crash** also enables you to examine the contents of the **putbuf** at any time, which is useful if your driver code is written to utilize this feature. You may want to use two terminals for debugging: one to monitor the driver with interactive **crash** and the other to issue commands that exercise the driver.

When you run **crash** on an active system, you access the */dev/mem* node, which is the default for the **-d** option. The command is

/etc/crash [-n */unix*] **[-w** *outputfile*]

You must use the kernel image that is running; if this is not named */unix*, specify the name of the file with the **-d** option. If you want the **crash** output to go to a file rather than to your terminal (standard output), use the **-w** option to specify the file. Note that the output of a specific **crash** function can be redirected to a file even if you do not use the **-w** in the **crash** command line.

Note that **crash** does not allow you to view active memory as it runs. Rather, you take an image of memory every time you issue a command and this is what you look at.

Using crash Functions

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

function [*argument*. . .]

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

Consult the **crash**(1M) page in the *System Administrator's Reference Manual* for a list of functions supported on your computer. Note that a number of **crash** functions from UNIX System V Release 2 were replaced with other functions on UNIX System V Release 3. Note also that, while most **crash** functions are common to all computers, each system also has unique functions that relate to specific devices supported on that machine. The **crash**(1M) manual page lists the valid **crash** commands.

Debugging with TRACE [3B400 Computer Only]

A TRACE driver allows you to look at a buffer in the crash dump to find out what the last few kernel events were. It is useful when debugging an internally complex driver. For instance, TRACE can help identify the cause of a deadlock condition for a driver that is handling communication protocols.

The UNIX operating system on the 3B15 and 3B4000 computers includes a TRACE driver as part of the basic system. Although this is part of the Virtual Protocol Machine (VPM) subsystem, you can use it for drivers that are not part of VPM as long as you obey the interface requirements. You will need to write a user program to interpret the output.

Using TRACE

The TRACE driver is described in **trace**(7) in the 3B4000 System Administrator's Manual. The procedure for using this tool is

1 Put many **trsave** function calls in your code. The calls are in the form

trsave(*dev, chno, buf, cnt*)
char *dev, chno, buf, cnt*;

where:

dev a minor device number for the trace driver

chno data stream channel number in the range of 0 to 15.

buf buffer containing the data for an event

cnt the number of characters in the buffer

An example of a **trsave** call is

```
trsave(0, 7, &entry, sizeof(entry));
```

Where "0" is the device number, "7" is the channel number, and "&entry" is the address of the buffer to be listed. In general, you can define this structure any way that is appropriate for your driver.

2 From user space, use **open**(2) to open the minor device number.

3 Then use **ioctl**(2) with the VPMSETC command to enable the selected channel.

Using the putbuf to Select Specific Channels

As an alternative to using the previously described **trace** driver, you can use the **putbuf** to select certain channels. To do this, use **cmn_err**(D3X) statements like the following in the driver code:

```
cmn_err(CE_NOTE, "!DEBUG: CH%, message, more message",channo);
```

The following **crash** command enables you to select only those messages for channel 4:

```
crash <<: ! grep 'CH4'
putbuf d a
. . .
```

Integration Testing

When you are satisfied with the performance of the driver in a fairly isolated environment, you should test the driver's functionality, error handling, and performance in an integrated environment. Activate as many other drivers on the system as possible, and do error-provoking tests as well as tests to ensure that the performance level remains adequate on an active system. As you will see later in this chapter, the interaction between drivers in a system may uncover errors that would never surface in tests run on an isolated driver. As a general rule of thumb, never ignore unexpected behavior on the system when you are testing the driver, particularly system level activity. For instance, watch for an increase in errors logged by other devices — your driver may be the cause.

Some examples of configurations on which the driver and the device should be tested are

- multiple copies of the new peripheral board in the system
- multiple subdevices on the new peripheral board
- various mixes of other peripherals, including those at the same or different bus request and interrupt priority levels
- (SBC-only) with and without VME memory boards present, using both block I/O and character I/O
- system heavily loaded with user processes (to ensure that pages are being allocated properly)

When testing a driver for an intelligent board, you may find it useful to use an emulator tool that enables you to start and stop the microprocessor used in that board.

ASSERT

ASSERT puts debugging code in the driver that checks for some condition that must be true. It panics the system if that condition is not true. This enables you to confirm that the kernel remains sane when your driver is installed.

To use ASSERT, include *debug.h* and compile the driver code with the "**-DDEBUG**" option to the **cc**(1) command.

The format for ASSERT is

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

ASSERT(expression);
```

ASSERT displays a message in the following format

PANIC: assertion failed: *expression*, **file:** *file*, **line:** *line#*

The message is also written to **putbuf**. ASSERT is defined in the */usr/include/debug.h* file.

An example is

```
35      ASSERT(mp != NULL);
```

If **mp** is equal to NULL, the system panics and displays

PANIC: assertion failed: mp != NULL, file: *file*, **line: 35**

Common Driver Problems

The next several pages discuss some of the common bugs in drivers with possible symptoms. These should be used only as suggestions. Each driver is unique and will have unique bugs.

Coding Problems

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

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

C Optimizer Bugs

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

Installation Problems

Installation problems refer to problems that prevent a system boot with your device configured. If the system won't boot, first try to boot it without the driver to verify that the driver is the problem. Chapter 5, "System and Driver Initialization," includes a list of driver rules that are enforced by the self-configuration process. Other driver problems that prevent a system boot are

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

  ```
  symbol undefined   set to zero
  ```

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

- Errors in the **init** or **start** routine. You can check that the initialization routine is being entered by inserting an unconditional **cmn_err** statement at the beginning of the routine.

- Allocating an array in the */etc/master.d* file then not declaring it as a global data structure for the driver or initializing it in an **init** or **start** routine. This will not prevent you from booting the system the first time, but may preclude a reboot from a */unix* file.

Data Structure Problems

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

Mismatched Data Element Sizes

Data element sizes in the master file should match those defined in the driver code. If the master file size is larger than the C-level definition, kernel memory is wasted but otherwise no harm is done. However, if the master file size is smaller than the C-level definition, the driver may overwrite some other driver's data when storing into what appears to be its own variable. This could cause the other driver to behave strangely, or might cause a kernel panic if it attempts to write beyond the mapped kernel memory.

To check this, use the **nm**(1) command to display the symbol table of the driver object file. For instance, if the header file includes

```
struct drv_struct  {
int      x;
short    y;
}
```

and the driver source code includes

```
struct drv_struct drv_xx
struct drv_struct drv_yy[10];
```

compile the code and examine the name list as follows:

```
$ cc -c -o drv.o drv.c

$ nm -x drv.o

Symbols from drv.o:

Name    Value       Class   Type   Size   Line   Section
drv.c               file
drv_xx  0x00000008  extern
drv_yy  0x00000050  extern

$
```

Because the value of an external variable in an object file is the number of bytes of storage it requires, the corresponding master file should define these elements as shown below. Note that the values of the columns other than DEPENDENCIES/VARIABLES are irrelevant for this discussion.

```
*    DRV driver
*
*    FLAG    #VEC    PREFIX    SOFT    #DEV    IPL    DEPENDENCIES/VARIABLES
-    cs      1       drv               1       4      drv_xx(0x8)
-    -       -       -         -       -       -      drv_yy[10](0x8)
                                or, as an alternative
-    -       -       -         -       -       -      drv_yy(0x50)
```

The above sequence works if the data items are defined and declared in the C code. The process is more complex if **lboot** is doing dynamic data definitions. For instance, if the driver code has

```
extern struct drv_struct drv_xx;      /* number of array elements varies
extern struct drv_struct drv_yy[];    /* dynamically at boot time

drv_sub()  {                          /* need to use externs or they disappear! */
 drv_xx.x++;
 drv+yy[0].y++
```

the compilation/name list session would yield the following:

```
$ cc -c -o drv.o drv.c

$ nm -x drv.o

Symbols from drv.o:

Name      Value       Class    Type      Size     Line   Section
drv.c                 file
drv_sub   0x00000000  extern   int( )    0x001e          .text
drv_xx    0x00000000  extern
drv_yy    0x00000000  extern


$
```

and the external variables are not flagged with any size.

To figure out what is needed in the master file, compile the driver with debugging information. Actually, during driver development, most compilations will include debugging information anyhow. Thus,

```
$ cc -g -c -o drv.o drv.c

$ nm -x drv.o

Symbols from drv.o:


Name        Value       Class     Type      Size      Line    Section
drv.c                   file
drv_struct              strtag    struct    0x0008
x           0x00000000  strmem    int                         (ABS)
y           0x00000004  strmem    short                       (ABS)
.eos                    endstr              0x0008            (ABS)
drv_sub     0x00000000  extern    int( )    0x001e            .text
.bf         0x00000009  fcn       int( )    0x001e    9       .text
.ef         0x00000017  fcn       int( )    0x001e    4       .text
drv_xx      0x00000000  extern
drv_yy      0x00000000  extern


$
```

This gives some excess information, and doesn't directly specify the size of **drv_xx** and **drv_yy**, but the size field of the **deg_struct** structure indicates the size of the element. To accurately communicate the size of the dynamic array, one more variable is required. So, the code becomes

```
extern struct drv_struct drv_xx;      /* number of array elements varies
extern struct drv_struct drv_yy[];    /* dynamically at boot time
extern int                            drv_cnt/* size of drv_yy */

drv_sub()  {                          /* need to use externs or they disappear! */
 drv_xx.x++;
 drv+yy[0].y++
```

The master file contains the following information:

```
*    DRV driver
*
*    FLAG    #VEC    PREFIX    SOFT    #DEV    IPL    DEPENDENCIES/VARIABLES
-    cs      1       drv               1       4      drv_xx(0x8)
-    -       -       -         -       -       -      drv_yy[10](0x8)
-    -       -       -         -       -       -      drv_cnt(%i) = { #C }
```

This dynamically allocates space for **drv_yy** according to the number of controllers present, and initializes **deg_cnt** to that number, so the C code can determine the size of the **drv_yy** array.

Value of Initialized Global Variables

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

Timing Errors

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

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

Improper IPL in Master File

If the IPL in the master file is not appropriate for this device on this system, the driver may cause system-wide data corruption or system sanity failure on a heavily-loaded system.

Corrupted Interrupt Stack

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

Referencing u_block Data Elements from Interrupt Level

The data elements of the u_block (see `user`(D4X)) should never be referenced from interrupt handling routines or subordinate routines that are called by these routines. This will cause random failure of processes on the system, frequently even processes that are not accessing this driver.

Accessing Critical Data

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

Overuse of Local Driver Storage

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

Incorrect DMA Address Mapping

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

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

Chapter 14: Performance Considerations

Contents

Introduction

One of the most important phases of driver development is evaluating the performance of the driver, which must include the overall impact a driver has on system performance. After a driver is written, tested, and debugged, adjustments may still be necessary to optimize performance and reliability. You may also want to create tools (or augment existing system tools) to monitor a driver's impact on system performance.

The first step in optimizing the performance of the driver is to run the kernel profiling tools (**profiler(1M)**) to identify where the driver spends the most time. Optimizing those areas will give the greatest gains in performance for the least effort. In most cases, these improvements can be accomplished by rewriting portions of C code.

If further performance enhancements are needed, some critical functions can be rewritten as **asm** pseudo-functions. The */usr/include/sys/inline.h* file defines a number of system functions (including **spl***) as **asm** macros. Including this header file in driver code may improve execution speed, but may also impact the portability of the driver to other UNIX System V processors or releases.

Using assembly language code in a driver will also make the driver more difficult to port and maintain. When converting C code to assembler to improve performance, be sure to comment out (rather than delete) the C code that provided the same functionality.

A driver with satisfactory performance may still degrade general system performance, either because it is monopolizes system resources or because the driver's tunable parameters are not set correctly. Integration testing of a driver, should include checking both resource usage and tunable parameters. Tools may be created to monitor the activity a driver, but be careful. Experienced programmers know that complex tools often create more system performance problems than they solve.

General Performance Guidelines

A number of general performance guidelines are summarized below.

1. Do not include extraneous code in the interrupt routines, but get in and out of these as quickly as possible.
2. Keep critical code sections (those that are protected by **spl***) as small as possible.
3. Choose **sleep** priorities that do not cause your driver to hog system resources.

Optimizing for Speed and Size

Optimizing code can mean either increasing execution speed, reducing the size of the code, or both. For driver code, "size" can refer to either the executable codesize or data size. Here the general term "driver object size" refers to the sum of code and the data size. Some optimization techniques will reduce both driver code and data size, while other techniques will trade off between them. Still other techniques will optimize for speed and the cost of driver object size.

The **size**(1) command can help to evaluate the driver object size, but it does not include any storage defined in the master file and allocated by self-configuration. For instance

```
size /boot/xdrv
5176 + 364 = 0 = 5540
```

does not include the variables defined in the master file:

```
xdrv_xdc[#C] (%0x29fc)
xdrv_cnt(%i) = {#C}
xdrv_spint[#C] (%0x08)
```

so the XDRV driver will need 0x3044 bytes of *.bss* and 4 bytes of *.data* per controller, in addition to the 5540 bytes that **size** lists.

Tools for Checking Driver Performance

Most driver performance improvements will come from analyzing how the driver works and looking for sections where it could be more efficient. The tools discussed in this section can be used to support this kind of analysis.

Testing I/O Operations for Block Devices

The system buffer cache header includes the **b_start** member, which can be used to monitor the amount of time required for an I/O operation. To use this, update the **b_start** member when updating other status information in the driver's **strategy**(D2X) routine, then write this value to the **putbuf** where it can be examined with the **crash**(1M) utility, as shown in Figure 14-1. Whenever measuring performance, write messages to **putbuf** to avoid the overhead of writing to the console.

The driver's interrupt handling routine will be called when the I/O transfer is completed. The **int** routine subtracts the value of **b_start** from the current time to determine the time required for the I/O transfer. The following code, from a disk driver, illustrates how this value is written to a queue that holds performance data, where it can be accessed for **sar**(1M) reports. Other options are to write it to a private queue that records performance data or to the **putbuf**.

```
dfstrategy(bp)
. . .
bp->b_start = lbolt;

#if TEST
    cmn_err(CE_NOTE, "!start time = %x\n",bp->b_start);
#endif

dfint(unit)
. . .
    dfcp->df_stat[drv].io_resp += (lbolt - bp->b_start);
```

Figure 14–1 Using b_start to Measure Block I/O Performance

Using the Disassembler to Analyze C Code

Disassembly involves "un-compiling" the object code to see what the compiler actually did with it. Driver code can be disassembled with either the **dis**(1) command or the **crash**(1M) **dis** function. In most cases, the **crash dis** function provides more useful information for analyzing driver code. Chapter 13 discusses how to use these tools.

Tuning the C Code for Performance

Significant performance improvements can often be realized by fine-tuning C code. Most application programming practices that enhance performance are also effective on driver code. These are well documented in the general industry literature, such as Jon Louis Bentley's *Writing Efficient Programs*.

In addition to algorithm analysis and code profiling, disassembling the C code and seeing what the compiler actually did with it may indicate areas that could be improved.

To use the code optimizer, the **cc** command line should include the **-O** option with the **-K sd** option (for speed optimization) or the **-K sz** option (for size optimization). The optimizer called by the **-O** option does not optimize assembler code or references to global variables.

The following sections discuss programming practices that may enhance the performance of C code.

Improving Both Speed and Code Size

In general, a shorter piece of code tends to run faster than a longer piece of code, although there are exceptions where a shorter piece of code might be slower, due to interactions with the instruction cache. Here are some suggestions that can be used to produce both smaller and faster code.

- Use local variables where possible (that is, when a variable is used only in one function and does not need to be global). Local variables can be addressed with shorter addressing modes and can be selected by the optimizer's register allocation algorithm to be placed into registers.
- In **for** loops that count from 0 to n, recode if possible so that counting is from n to 0 (so that the loop termination condition is a test against zero).
- Use integers in place of **char** and **short** variables unless the variables are in an array or an array of structures.
- Use integers or characters in place of bit fields unless the bit fields are in an array or an array of structures.

- Put frequently used, inner block local variables and procedure arguments into registers. If you know which variables are used frequently at run-time, you can complement the optimizer's register allocation algorithm by declaring frequently used variables as registers. The following example shows how this technique can be used:

```
msg_process(type,msg_ptr)
int type;
char *msg_ptr;
{
   if(type == MSG)
   {
      register char *cp;
      for(cp = msg_ptr;*cp;cp++)
      .
      .
      .
   }
   else . . .
}
```

  In this example, the variable *cp* is explicitly defined as a register variable.

- Replace array indexing operations with pointer operations. As an example, the array indexing operations

```
int matrix[50];
int i;
for(i=0; i < 50; i++)
      matrix[i]=0;
```

  can be transformed into the pointer operations

```
int matrix[50];
int *ip;
for (ip=matrix; ip <= &matrix[49];ip++)
                    *ip=0;
```

  to increase execution speed and reduce code size. This array will be in the kernel's *.bss*, if it is external to the function. Since for a driver this runs only once during system initialization, the performance impact is minimal.

- Replace frequent references to global data structures by a local pointer which can be optimized into a register. For instance, consider code that frequently writes to the `u_block`:

```
routine(...)
                    {
                    ...
                    u.arg = ...
                    u.otherarg = ...
                    ...
                    }
```

  Performance may be improved when the above example is rewritten to include a local pointer to the `u_block`:

```
routine(...)
                    {
                    register struct user * uptr = &u;
                    ...
                    uptr->arg = ...
                    uptr->otherarg = ...
                    ...
                    }
```

Increasing Speed

The following recommendations may help to increase code execution speed, although driver object size may be increased.

- Use the **-O -K sd** options on the **cc** command line.
- Put small routines in the same file as the routines calling them. The small routines can then be expanded in-line by the optimizer.
- Use short integers or characters in place of bit fields, even in arrays or arrays of structures.
- Use signed in place of unsigned integers, unless the higher numeric range of unsigned values is required.

- Some low repetition loops (less than three iterations) can be unrolled into straight-line code to decrease the loop indexing overhead. For example

```
for(sum=0,i=0;i<=2;i++)
                sum += X[i];
```

can be replaced with

```
sum = X[0] + X[1]] + X[2];
```

Unroll the loop only if the unrolled loop is smaller than 256 bytes or if the original loop is already larger than 256 bytes, (size of the instruction cache). While this will improve performance, it may make the driver code harder to read and maintain. Be sure to provide adequate comments.

Reducing Driver Object Size

The following techniques can be used to reduce the size of object code, possibly at the expense of execution speed.

- Use the **-O -K sz** options on the **cc** command line.
- Use characters or short integers in place of integers within arrays and structures. In the case of structures, care must be taken in the ordering of structure members so that alignment requirements (for example, shorts on halfword boundary) do not negate potential savings from the smaller data size by creating holes.

Example of Improved C Code

This example shows how some careful reworking of the C code can significantly improve performance. Figure 14-2 is a simplified version of the **read**(D2X) routine for a network driver before it was reworked. A performance analysis tool measured the receive throughput for the driver at 5966 characters per second (cps). The **read** routine contains statements that are executed once-per-64 characters, once-per-16 characters, and once-per-character. Because they are executed most often, the once-per-character statement should be examined most closely.

Note the definition of the first two variables (lines 3 and 4). The compiler being used allows only one pointer register variable and one register variable for a short integer or character. These register variables are taken as the first two variables defined in a function. Placing the most frequently used variables in registers improves the performance of the driver.

```
1  pre_read()
2  {
3                                   register unsigned char *ptr;/* MUST BE FIRST */
4                                   register short _fib;/* MUST BE FIRST */
5                                   unsigned short c;
6                                   struct pre_pkbufr *pkb;
7                                   unsigned short bitloc = 0100000;
8
9                                   /*
10                                  ** WHILE not empty
11                                  */
12                                  while ( !(inw(STATUS) & RCVR_EMPTY) ) {
13                                  MOV_I(c);
14
15                                  /*
16                                  ** WHILE not empty AND no frame or parity error
17                                  **   AND char is channel number
18                                  */
19                                  while ( !(inw(STATUS) & RCVR_EMPTY) ) {
20                                  if ((c & (PARITY_ERR | FRAME_ERR))) {
21                                  if (c & PARITY_ERR) {
22                                  stats.parity_err++;
23                                  }
24                                  if (c & FRAME_ERR) {
25                                  stats.frame_err++;
26                                  }
27                                  break;
28                                  }
29                                  if ( !(c & CHAN_NUM) ) { /* keep looking for chan */
30                                  break;
31                                  }
32                                  ptr = &pkb->Pdata[0];
33
34                                  /*
35                                  ** WHILE not empty AND not a channel number
36                                  */
37                                  while ( !(inw(STATUS) & RCVR_EMPTY) ) {
38                                  MOV_I(c);
39                                  if (c & (PARITY_ERR | FRAME_ERR)) { /* parity/frame error */
40                                  continue;
41                                  }
42                                  if (c & CHAN_NUM) {/* it's a channel number */
43                                  break;
44                                  }
45
46                                  switch (c = c & MASK1) {
47                                  /* Protocol control characters */
48                                  case P_C_0:
49                                  case P_C_1:
50                                  case P_C_2:
51                                  ...
52                                  case P_C_n:
53                                  default:
54                                  if (c & DATA_CHAR) { /* we got data */
55                                  if (pre_p->tail) { } /* trailer started? */
56                                  else { /* just data */
57                                  *ptr++ = c & CHAR_MASK;
58                                  pkb->Plen++;
59                                  bitloc >>= 1;
60                                  }
61                                  break;
62                                  }
```

```
63
64                              /* more frequent protocol control characters */
65                              if (((c & MASK2) == P_C_x0) || ((c & MASK2) == P_C_x1)) { }
66                              if ((c & MASK2) == P_C_x2) { }
67                              if ((c & MASK3) == P_C_x3) { }
68                              if ((c & MASK2) == P_C_x4) { }
69                              if (c & SUPERVIS) { } /* supervisory control */
70                              else { /* in-line control character */
71                              *ptr++ = c & CHAR_MASK;
72                              pkb->Plen++;
73                              pkb->Phibits |= bitloc;
74                              bitloc >>= 1;
75                              }
76                              }/* end of switch on 'c' */
77                              } /* not empty AND not channel number */
78                              }/* not empty AND channel number */
79                              }/* not empty */
80 }
```

Figure 14–2 read Routine Before Being Improved

The body of the **pre_read** routine contains three nested loops. The outermost loop reads characters from the receive FIFO into the variable *c*. The middle loop searches for a channel number (signified by the CHAN_NUM bit being set). This loop does not read characters. It is always entered at the top with a character in *c*. This character comes either from the outermost loop or from breaking out of the innermost loop when a channel number is found. The innermost loop processes the packet contents. For each character, the character type is determined and appropriate actions taken.

In lines 38 – 49 the code first checks that the character received is data, then checks for a number of other conditions. Less frequently encountered protocol control characters are checked for before the more frequent control characters. Unlike many compilers, the one being used implements the switch as a series of test-and-jumps. Figure 14-3 shows how this innermost loop was rewritten, increasing receive throughput to 7071 cps.

```
1                   /*
2                   ** WHILE not empty AND not a channel number
3                   */
4                   while ( !(inw(STATUS) & RCVR_EMPTY) ) {
5                   MOV_I(c);
6                   if (c & (PARITY_ERR | FRAME_ERR)) { /* parity/frame error */
7                   continue;
8                   }
9                   if (c & CHAN_NUM) {/* it's a channel number */
10                  break;
11                  }
12                  c &= MASK1;
13                  if (c & DATA_CHAR) { /* data rather than control */
14                  if (pre_p->tail) { } /* trailer started? */
15                  else { /* just data */
16                  *ptr++ = c & 0377;
17                  pkb->Plen++;
18                  bitloc >>= 1;
19                  }
20                  break;
21                  }
22
23                  /* more frequent protocol control characters */
24                  switch (c & MASK2) {
25                  case P_C_x0:
26                  case P_C_x1:
27                  case P_C_x2:
28                  case P_C_x4:
29                  default:
30                  }
31
32                  if ((c & MASK3) == P_C_x3) { }
33                  if (c & SUPERVIS) { } /* handle supervisory control */
34                  else if (c & INLINE) { /* in-line control character */
35                  *ptr++ = c & 0377;
36                  pkb->Plen++;
37                  pkb->Phibits |= bitloc;
38                  bitloc >>= 1;
39                  }
40
41                  /* less frequent protocol control characters */
42                  switch (c) {
43                  case P_C_0:
44                  case P_C_1:
45                  case P_C_2:
46                  ...
```

```
47                      case P_C_n:
48                      default:
49                      }
```

Figure 14–3 Rewritten Innermost Loop for pre_read

Rather than using interrupts, this driver has statements in the two inner loops that check for frame or parity errors. By removing these, (lines 6-11) throughput increased to 7282 cps.

Next the developers looked at the "character is data" case within the innermost loop. The sole short register variable was being wasted by the implementation of **MOV_I(A)**. The macro was changed to leave the 16-bit word in *_val* (which resides in the **cx** register of the 80186 microprocessor) rather than moving it to a passed argument. The macro, now called **MOV_I_VAL( )** had two advantages over its predecessor.

1. The new macro could be implemented with fewer instructions, since a final "move" to the passed argument was no longer required.

2. The 16-bit word in *_val* could now be used in computations. Previously, the stack variable *c* had been used for computation.

All references to *c* were changed to *_val*, making the most critical variable in the routine a register variable. The throughput increased to 7816 cps.

The innermost loop of the routine is now reading the next character, checking for a channel number, masking, then checking the "character is data" case.

When processing a data character, only the lower 8 bits of the character were used. This made the masking done before the "character is data" check redundant if the character was indeed data. By moving the "masking" statement from before the "character is data" check to after the check, throughput increased to 8309 cps.

Next, the variable *bitloc* (line 18) was removed from the routine. Since in-line control characters were rare events, driver performance was improved by having the driver calculate *bitloc* when it was needed, thus eliminating another statement from the frequently-used "just data" case.

Another change to the "just data" case was to remove the masking off of the upper byte of *_val* before the character was put into the packet buffer. This modification was also made when handling in-line control characters. Disassembling the code showed that the statement

```
*ptr++ = _val & 0377
```

was turned into assembly instructions which performed the logical AND operation on *_val*, then put

the lower half of the **cx** register (*_val*) into the buffer. Casting *_val* to an unsigned character had the same effect, eliminating the logical AND instruction. So, the statement was changed to

```
*ptr++=(unsigned char) _val;
```

With these two modifications improved, throughput increased to 8503 cps. Figure 14-4 shows the improved **read** routine.

```
1   pre_read()
2   {
3   register unsigned char *ptr;/* MUST BE FIRST */
4   register short val;/* MUST BE FIRST */
5   struct pre_pkbufr *pkb;
6
7   /*
8   ** WHILE not empty
9   */
10  while ( !(inw(STATUS) & RCVR_EMPTY) ) {
11  MOV_I_VAL();
12
13  /*
14  ** WHILE not empty AND char is channel number
15  */
16  while ( !(inw(STATUS) & RCVR_EMPTY) ) {
17  ...
18  ptr = &pkb->Pdata[0];
19
20  /*
21  ** WHILE not empty AND not a channel number
22  */
23  while ( !(inw(STATUS) & RCVR_EMPTY) ) {
24  MOV_I_VAL();
25  if (val & CHAN_NUM) {/* it's a channel number */
26  break;
27   }
28  if (val & DATA_CHAR) { /* data rather than control */
29  if (pre_p->tail) { } /* trailer started? */
30  else { /* just data */
31  *ptr++ = (unsigned char) val;
32  pkb->Plen++;
33  }
34  break;
35  }
36
37  /* more frequent protocol control characters */
38  ...
39  else if (val & INLINE) { /* in-line control character */
40  *ptr++ = (unsigned char) val;
```

```
41                      pkb->Phibits |= (0100000 >> pkb->Plen);
42                      pkb->Plen++;
43                      }
44                      ...
45                      } /* not empty AND not channel number */
46                      }/* not empty AND channel number */
47                      }/* not empty */
48}
```

Figure 14–4 Improved pre_read Routine

Using Assembly Language in Driver Code

If rewriting the C code does not give you acceptable performance for the driver, you may want to rewrite the critical sections in assembler. If you only need to write a small piece of a routine in assembler, you can use an **asm** escape from C. In general, however, the **asm** escapes are hard to maintain and you should write **asm** pseudo-functions for the appropriate sections.

Writing asm Pseudo-functions

The **asm** facility lets you define constructs that look like static C functions and can access C symbols. Each **asm** macro has one definition and zero or more uses per source file. The definition must appear in the same file as its use or be included in that file; the same **asm** macro can be defined differently in different files for one driver.

The body of an **asm** pseudo-function contains lines specifying possible storage classes of the arguments. Each storage specification line is followed by lines of text into which the pseudo-function call will be expanded if the storage class specification line matches the actual arguments.

The **asm** macro definition declares a return type for the macro code, specifies patterns for the formal patterns, and provides bodies of code to expand when the patterns match.

As the **cc** compiler expands the code body, it replaces each formal parameter in an **asm** macro with its idea of the assembly language locations of the actual arguments.

When used, **asm** macros look like normal C function calls. The can be used in expressions and can return values. The arguments to an **asm** macro can be arbitrary expressions, as long as they do not contain uses of the same or other **asm** macros.

When the argument to an **asm** macro is a function name or structure, the compiler generates code to compute a pointer to the structure or function; the resulting pointer is used as the actual argument of the macro.

If the **asm** definition and the **asm** use differ in number of parameters, the compiler silently generates a normal subroutine call. This may lead to an unresolved external reference.

The **asm** body is processed by the C preprocessor. C-style comments (prefaced by /*) are removed at that time. The C preprocessor recognizes conditional blocks (**#if**, **#ifdef**, and **#ifndef** constructs) that are contained within an **asm** macro.

A **#ident** statement in an **asm** macro will be ignored by both **as** and **cc**. As expected, a **.ident** pseudo-op used within an **asm** macro produces a *.comment* section in the *.o* file.

Definition of asm

The syntactic descriptions that follow are presented in the style of *The C Programming Language* by Brian Kernighan and Dennis Ritchie. The syntactic classes type-specifier, identifier, and parameter-list have the same form as in that document. Elements enclosed in square brackets "[ ]" are optional, unless the right bracket is followed by "+", which means "one or more repetitions" of a description. Similarly, "*" means "zero or more repetitions."

asm macro:

asm [*type-specifier*] *identifier* ([*parameter-list*])
{
[*storage-mode-specification-line*
asm-body] +
}

That is, an **asm** macro consists of the keyword **asm**, followed by what looks like a C function declaration. Inside the macro body there are one or more pairs of *storage-mode-specification-line* (pattern) and corresponding **asm-body**. If the type-specifier is other than *void*, the **asm** macro should return a value of the declared type.

storage-mode-specification-line:
% [*storage-mode* [*identifier* [, *identifier*]*] ;]+

That is, a *storage-mode-specification-line* consists of a single line (no continuation with \) that begins with % and contains the names (identifiers) and storage modes of the formal parameters. Modes for all formal parameters must be given in each *storage-mode-specification-line* (except for *error*). Both the % and the terminating "}" must be the first character on that line. If an **asm** macro has no parameter-list, the *storage-mode-specification-line* can be omitted.

The compiler recognizes the following storage modes in **asm** macros:

treg A compiler-selected temporary register.

ureg A C register variable that the compiler has allocated in a machine register.

reg A **treg** or **ureg**.

con A compile-time constant.

mem An operand that matches any allowed machine addressing mode, including **reg** and **con**.

lab A new label. The identifier(s) that are specified as being of mode **lab** do not appear as formal parameters in the **asm** macro definition, unlike the preceding modes. Such identifiers must be unique.

error Generate a compiler error. This mode exists to allow the programmer to flag errors at compile time if no appropriate pattern exists for a set of actual arguments.

The **asm** body represents assembly code that the compiler generates when the modes for all of the formal parameters match the associated pattern. Syntactically, the **asm** body consists of the text between two pattern lines (that begin with "%") or between the last pattern line and the } that ends the **asm** macro. C language comment lines are not recognized as such in the **asm** body. Instead they are simply considered part of the text to be expanded.

Formal parameter names can appear in any context in the **asm** body, delimited by non-alphanumeric characters. For each instance of a formal parameter in the **asm** body the compiler substitutes the appropriate assembly language operand syntax that will access the actual argument at run-time. As an example, if one of the actual arguments to an **asm** macro is **x**, an automatic variable, a string like **4(%fp)** would be substituted for occurrences of the corresponding formal parameter. An important consequence of this macro substitution behavior is that **asm** macros can change the value of their arguments. Note that this is different from standard C semantics.

For **lab** identifiers, a unique label is chosen for each new expansion.

If an **asm** macro is declared to return a value, it must be coded to return a value of the proper type in the machine register that is appropriate for the implementation.

No line within the **asm** body can start with "%" or "$".

Optimizing Code Containing asm

The **-O** option to the **cc** command optimizes all code in a function except the **asm** code.

An **asm** must conform to the following restrictions if the surrounding code is to be optimized:

- The **asm** cannot contain a branch to or from another **asm** or any other point in the program outside the body of the **asm** itself. Function calls are permitted within the **asm**, and it is not required that the called function return. Except for functions that do not return, control following execution must fall through to the next executable statement.
- The **asm** should not modify code generated by the compiler or affect the contents of registers on which the generated code depends. It might change the contents of scratch registers (%r0 through %r2) but should not modify user registers (%r3 through %r8).

It is the programmer's responsibility to ensure that code containing **asm** works correctly when optimized.

How to Use asm

This example shows how to define and use **asm** macros. Two macros are defined: **spl7** and **splx**. The **spl7** macro changes the priority to the highest possible level; the **splx** macro restores the priority to its previous level.

The definition of **spl7** is:

```
asm     int
spl7( )
{
        MOVW    %psw,%r0
        MOVW    &0x1e100,%psw   #mask all interrupts
}
```

The definition of **splx** is:

```
asm     int
splx(opsw)
{
%       mem     opsw;
        MOVW    %psw,%r0
        MOVW    opsw,%psw
%       reg     opsw;
        MOVW    %psw,%r0
        MOVW    opsw,%psw
}
```

An example of the use of these macros is:

```
untimeout(untid)
register untid;
{
   register struct callo *p1, *p2;
   register s;

   s = spl7( )
   / * protected code * /
   splx(s);
}
```

Drivers and System Performance

In addition to optimizing the performance of your driver, you need to ensure that your driver is not degrading system performance. To do this, you will need to monitor system performance with your driver active on a live system. Factors in your driver to check include the following:

- Intense buffer use in your driver may reduce performance of other drivers or user processes because of the reduced memory available on the system.
- Sleep priorities that are set too high may be causing your driver to unnecessarily "hog" system resources.
- Some system tunable parameters may need to be modified because of the presence of the new driver.

Using System Buffers

Whether the driver is using a standard or private buffering scheme, avoid consuming a disproportionate amount of system resources. The following practices are suggested:

- Be sure to release buffers when they are no longer needed (**brelse**(D3X) and **putcf**(D3X) functions).
- The kernel tunable parameters NBUF (for system buffers) or NCLISTS (for `cblocks`) may need to be modified because of your driver.

Checking Sleep Priorities

Chapter 9 discussed how to determine sleep priorities levels and whether or not the process should ignore the receipt of signals.

Driver Impact on System Tunable Parameters

The */etc/master.d/kernel* file contains several system tunable parameters that may need to be modified to accommodate a new driver. The administrative documentation describes these in more detail. This section only discusses the impact a new device may have on tunable parameters.

NCLIST Specifies the number of `cblocks` to allocate to the `cfreelist` structure. If the new character device(s) use `clists` for buffering, this parameter should be increased. The general rule is to allocate eight buffers for each device that is using `clists`.

NBUF Specifies the number of system buffers to be allocated to the system buffer cache. This number may need to be increased for a new block-access device.

NHBUF Specifies the number of "hash buckets" to allocate in the system buffer cache. This value must be a power of 2 and should be equal to NBUF.

NINODE Specifies the number of *inode* table entries to allocate. If the driver being installed significantly increases the number of files that will be opened at a given time, this number may need to be increased.

NFILE Specifies the number of open file table entries to allocate. This number should be slightly less than NINODE; if NINODE is increased, NFILE should also be increased.

Chapter 15: Porting Drivers

Contents

Introduction

Porting a device driver to another machine can be difficult, because drivers are more sensitive to machine-specific details than other software. This chapter discusses problems likely to be encountered when porting between systems supported by this manual. It shows how to isolate machine-dependent sections of code, and gives guidelines for porting drivers from other UNIX System releases and machines.

Although object-code portability for drivers is not feasible at this time, many drivers can be ported by merely recompiling their source code on the new system. When the driver is recompiled, it picks up much system-specific information from the header files. For instance, while there are some differences in the `user` structure between machines, the *sys/user.h* header file always defines the structure as it is implemented on that machine.

For more information about porting drivers, see J. E. Lapin's *Portable C and UNIX System Programming*. It explains the relationships between the various UNIX dialects, points out common pitfalls when porting code, and provides some helpful insight into writing portable C code. Of particular interest is the section describing a portable interface to the version-dependent features of TTY drivers.

Making Driver Code Portable

For a number of reasons, some sections of most driver code is not totally portable. The following sections discuss methods for writing driver code that isolates non-portable code sections.

Using Conditional Compilation Statements

Conditionally compiled statements are useful when only a few sections of the driver code, master files, and header files are non-portable. However, if used excessively, they can make the code difficult to read and maintain.

Driver code and header files use the standard C compiler conditional statements, primarily **#if**. The **-D** directive to the C preprocessor (called by **cc**(1)) lets you specify the version- or machine-specific code that should be included or excluded. The two left columns in Table 15-1 give the system definitions that are recognized by the preprocessor; the two right columns give the conventional system definitions for 3B4000 adjuncts, which must be defined to the C compiler.

Table 15−1 C Preprocessor System Definitions

Definition	*System*	*Definition*	*System*
u3b2	Any 3B2 computer or SBC	u3badp	3B4000 ADP kernel
u32100vme	SBC computer	u3badp	3B4000 EADP kernel
u3b15 or HOST	3B15 or 3B4000 Master Processor	u3bacp	3B4000 ACP kernel
u3b	3B20 computer	ADJUNCT	any 3B4000 adjunct (ACP, ADP, or EADP)
vax	DEC® VAX system		
pdp11	DEC PDP-11 system		

Double OR bars are used to indicate an alternative system. For instance, if you have code that should run for the 3B2, the 3B15, or the 3B4000 ACP kernels, the syntax is:

```
#if u3b2 || u3b15 || u3bacp
    code
#endif      /*u3b2 || u3b15 || u3bacp */
```

The conditional statements can also be used to specify a section of code that should not be included for a specific system. For example:

```
#if !(u3badp | u3beadp)
```

is interpreted to mean "if neither u3badp or u3beadp." The **#ifndef** statement has a similar meaning, so:

```
#ifndef u3b2
```

means "if u3b2 is not defined", or "do this on any kernel other than u3b2."

The following syntax is also legal:

```
#if !defined(u3b15) && !defined(u3b2)
```

meaning "if neither u3b15 nor u3b2 is defined, do this."

All conditionally compiled sections of code must be terminated with a **#endif** statement; this line should be commented to indicate the condition being closed, as in the example above.

Writing Machine-Dependent Subroutines

When a driver must have large portions of machine-dependent code it should be isolated in separate routines. The conditional statements can then be used to call the appropriate subroutine for the system. This is the recommended approach, for example, for isolating code that must interact directly with the 3B4000/3B15 dual-MMU.

Porting Drivers from Other Systems

This section lists some of the modifications that may be necessary when porting drivers from other hardware, other versions of the UNIX operating system, or UNIX System V Release 2. This list is not exhaustive, but provides information on some known porting problems.

printf Driver Function

Earlier UNIX releases used the **printf** kernel function to send driver messages to the console. The kernel's **printf**(3S) should be replaced (in UNIX System V, Release 3) with the **cmn_err**(D3X) function.

panic Driver Function

In other UNIX system releases, BCI drivers used the **panic** kernel function to send a message to the console and panic the system. The proper convention in UNIX System V Release 3 is to use **cmn_err**(D3X) with the "CE_PANIC" argument. For example

```
panic("shminit: tunable parameter PREGPP too small for shared memory\n");
```

should be replaced with:

```
cmn_err(CD_PANIC,"shminit: tunable parameter PREGPP too small for shared memory");
```

Conditional Preprocessor Statements

In UNIX System V C Programming Language Utilities (CPLU) Release 3.1 and forward, the preprocessor requires a matching **#endif** statement for all **#if**, **#ifdef**, and **#ifndef** statements. If a **#endif** is omitted, the compiler gives the following error message:

```
Unexpected EOF within #if, #ifdef, or #ifndef
```

With the use of **#include** statements, the **#endif** statement can be in a file other than the initial conditional statements, although driver code is easier to maintain when the conditional statements and terminators are in the same file.

Labels on **#endif** statements may produce warnings during compilation, which may be ignored.

Machine-Specific Function and Structure Information

This section discusses function and structure differences that may impact driver portability among the machines supported by this book.

Machine-Specific Functions

Table 15-2 lists the Section D3X functions that are supported on some but not all computers covered in this document.

Table 15–2 Machine-Specific Functions

	Computer		
Function	*SBC*	*3B2*	*3B15/3B4000*
getvec		X	
dma_breakup	X		
drv_rfile			X
getsrama			X
getsramb			X

IPL-to-spl Correspondence

As the table on the **spl***n*(D3X) reference page shows, the IPL-to-spl correspondence varies between machines. When porting hardware and the associated drivers, it may be necessary to modify the **spl** numbers or the IPL of the device to ensure that critical code sections run at the proper execution level.

MMU Implications for Porting

Chapter 6 discusses the dual-MMUs used on the 3B15 computer and 3B4000 Master Processor. Many drivers will not require special coding for the dual MMUs, as long as the driver is compiled using the 3B15 header files. Drivers that extract a section id from a virtual address or reference SRAMs as simple arrays will have to be recoded to utilize the dual MMUs, as will drivers that do virtual-to-physical translation, although the impact on drivers that use the **vtop**(D3X) function will be less than on those that have their own software translation routines.

In conjunction with driver changes, any corresponding intelligent device firmware must be analyzed for possible dual MMU impacts. When firmware accesses memory management tables or relies upon a breakdown of a virtual address to translate addresses, the rules and assumptions made must be carefully examined. Data passed from software to firmware for use in address translation must be coordinated. In some cases, a choice can be made as to whether firmware will be changed or whether the corresponding software driver will accommodate the dual MMU changes. For example, the driver for the IDFC disk controller on the 3B15 computer is passed SRAMA and SRAMB values and performs its own virtual-to-physical translations. The firmware, which was originally designed to run on a single MMU computer, uses bit 29 as part of the SSL (Segment Select field). Rather than change the firmware to ignore bit 29, the IDFC driver departs from the standard use of the **getsrama**(D3X) function and passes unadjusted SRAMA/SRAMB values so that using bit 29 will still result in the correct address translation.

Chapter 16: Packaging the Driver

Contents

Introduction

This chapter gives instructions for packaging the driver software for resale and installation on other systems.

All software packaged for any of the systems covered in this book must include INSTALL and DEINSTALL scripts that run under the system administration utility (**sysadm**(1M)). Detailed instructions on writing these scripts are in the *Application Software Packaging Guide*. See Chapter 1 for information on how to order this document.

Items to Check Before Running INSTALL

INSTALL scripts for drivers should check for the following conditions before proceeding to install the driver on the system:

1. this driver has not already been installed
2. no file in the */etc/master.d* directory uses the same prefix as this driver
3. all dependencies of this driver are honored
4. files associated with this driver do not have the same name as any existing files on the system. Check the */usr/include/sys*, */etc/master.d*, */boot*, and appropriate */usr/src/uts/io* subdirectories.

Such checks are more necessary for drivers than for most other software, since driver software and associated files must go into certain specified directories.

Installation Steps

Chapter 12 discusses the general steps for installing a driver.
The following list describes how and when these should be performed in relationship to the system administration INSTALL script:

I The following should be complete *before* running INSTALL:

- Install the hardware on the system.

II The following functions should be performed by the INSTALL script:

- Confirm that this driver is not already installed.
- Check that all dependencies of this driver are met.
- Check that the space requirements for this driver are met.
- Create any */etc/passwd* or */etc/group* entries that may be required for software related to this driver.
- Create the header file(s) in */usr/include/sys* or appropriate subdirectories.
- If you are releasing driver source code, create the source code files in the *io*, *master.d*, and *sys* subdirectories of the */usr/add-on/DRIVER-NAME* directory.
- Compile the object file in the same directory as the source code.
- Create the master file in the */etc/master.d* directory
- For software drivers, generate a major number in the master file and create the bootable object file in the *boot* directory using the **drvinstall**(1M) command.
- For hardware drivers, generate the bootable object file in the *boot* directory using the **mkboot**(1M) command.
- On systems other than the 3B15 and 3B4000 MP, create the diagnostics package in */dgn* and add the driver to the *edt_data* table with the **edittbl**(1M) command.
- On the SBC and 3B2 computers, set up scripts that create special device files in either the */etc/brc.d* or */etc/rc.d* directory (for devices other than Disk or Serial). Use the **getmajor**(1M) command to get the external major number for these scripts. On the 3B15 and 3B4000 computers, create the special device files under the */dev* directory.

- ☐ If required, install pumpcode for this device.
- ☐ Install the **edtgen** utility.
- ☐ Create the package tracking file(s) in the */usr/options/xxx* directory.

III The following activities should be done manually *after* running INSTALL:

- ☐ Make a backup copy of the */unix* file.
- ☐ Shutdown and reboot the system.
- ☐ If necessary, adjust the values of kernel tunable parameters that may be affected by the presence of the driver.

The UNINSTALL script can do all deinstallation steps listed in Chapter 12, except for physically removing installed hardware.

The Driver Update Package

A driver update package is installed on top of an existing driver package to correct errors or enhance capabilities of the driver.

The INSTALL script for an updated software driver or loadable module must

- use the major number already assigned to the */etc/master.d* file
- accept the object, master, and system files and creates a driver image for use with "driver add at boot" (using the **mkboot** command)
- edit the */etc/system* file, removing the old INCLUDE line and replacing it with the new INCLUDE line[1]

The INSTALL script for an updated hardware driver accepts the object, master, and system files and creates a driver image for use with "driver add at boot" (using the **mkboot** command). The major numbers for hardware drivers are assigned by the **getmajor** utility. The board address is used as the major number in the */etc/master.d* file. Hardware drivers are automatically self-configured if a board is plugged into the system at boot time. Customers should be told to add an EXCLUDE line manually to the */etc/system* file if they want to boot the system with the hardware board and not include the driver image in the configuration.

1. The **drvinstall**(1M) command does this for software drivers.

Appendix A: Equipped Device Table (EDT)

Contents

Appendix A: Equipped Device Table (EDT)

This appendix describes the equipped device table (EDT) for the Single Board Computer (SBC), the 3B2 computers, and the 3B15 and 3B4000 computers.

The EDT is a table in the private memory associated with the CPU that lists all hardware devices present on the system (except memory cards/boards). Self-configuration configures all devices listed here, unless they are specifically listed in an EXCLUDE line in the */etc/system* file or if there is no driver in the */boot* directory.

When a SBC, 3B2 computer, or 3B4000 ACP is brought up, the computer firmware builds a skeleton EDT. The firmware then calls **filledt**(8), which accesses the *edt_data* file and populates the EDT in memory. The *edt_data* file is in the */dgn* directory on the SBC and 3B2 computers, and in the */adj/pe#/dgn* directory on an ACP. (# is the Processing Element (PE) number.)

When a 3B15 computer or a 3B4000 Master Processor (MP) is brought up, the EDT is built by the initialization software from *edt_data* files that are kept for the MP, the 3B4000 ACP, and the Small Computer System Interface (SCSI) bus. Extended EDTs are built on intelligent controllers by the controller firmware, such as the SCSI Local Bus Interface Circuit (SLIC). The extended EDTs exist in the memory of the controller.

SBC EDT Architecture

The UNIX system firmware on the SBC was developed from that on the 3B2/400 computer and was kept as similar to it as possible. The SBC has no slots and devices can be placed at any physical address as long as no two are at the same address. To continue using the same mechanism as the 3B2/400 for system configuration, the concept of slots was replaced by an index into the EDT table. Consequently, device drivers get their addresses from tables. Interrupt vectors and external major device numbers are still derived from slots and **lboot** still uses the presence of a device in the EDT to decide whether to include the corresponding device driver when linking a UNIX system kernel.

Because SBC peripherals do not contain ROMs with WE 32100 microprocessor code for firmware execution and the system boot, this code must be compiled into the firmware for boot devices. A mechanism was added to the firmware so that the boot device can be discovered before booting. Other devices can be added by **filledt**(8) later.

3B2 Computer or 3B4000 ACP EDT Architecture

The 3B2 computer (or 3B4000 ACP) has I/O slots with predetermined addresses into which peripheral boards may be plugged for I/O devices. The boards appear in the CPU's physical address space at known addresses (determined by the slot in which they are located). Each board has a read-only register that defines what kind of board it is.

When the firmware is initialized, the computer probes all the slots and puts information from the ROM on each board in the EDT in main memory. EDT information includes such things as whether the device can be a boot device, whether it can be a system console, or if it requires that firmware be loaded before operation. The system console and integral floppy and hard disks are treated as controllers for device #0. The slot number is used for such things as determining the device's external major number and calculating the device's physical address and interrupt vector(s).

When the system is powered up, it runs **filledt**(8). The **filledt** process uses information in the */dgn/edt_data* file (*/adj/pe#/dgn* on the ACP) to add further information to the EDT tables in memory, including the subdevices attached to each controller. The diagnostic program, **dgmon**(8), uses this information to load and run diagnostic packages from the system disk. The system booter/linker (**lboot**) uses the EDT tables to decide which device drivers should be linked into the kernel and which external major device numbers should be used for them.

The 3B2 500/600 computers differ from the 3B2 300/400 computers in these ways

- BUBUS — or BUffered micro BUS, a bus designed for handling devices external to the main bus. The inclusion of this bus does not affect driver development and is mentioned here only as a reference. When the EDT is displayed from firmware, the BUBUS is displayed as either the "buffered microbus" or the "microbus."
- **cons_cap** and **cons_file** fields — not used. These fields in the EDT indicate the device's use of the console. However, when inserting an entry into the *edt_data* file, you are still prompted to enter information for these fields. These prompts are maintained for downward compatibility among members of the 3B2 computer family.
- **word_size** — has a different meaning. In the past, this one-bit wide field designated that the word size would be either 8 bits (0) or 16 bits (1). With the advent of the 32-bit word sizes required by some of the interfaces built-in to the 3B2 500/600 computers, this field came to have a different meaning. The 0 value still means an 8-bit word size, but the 1 now indicates that the word size is at least 16 bits. The exact word size can only be found by using the **edt** command in firmware mode, or the **show** command with the diagnostics monitor, DGMON. In these EDT listings, the word size is found under the "word width" notation expressed in bytes.

Finally, the 3B4000 ACP differs from all other 3B2 computers in that it does not have its own console. Therefore, commands that interact with firmware cannot be invoked on the ACP. Instead, the ACP uses a command shared with the 3B4000 MP and 3B15 computers to display the contents of the EDT.

Displaying the EDT

The EDT can be displayed in a variety of ways depending on the type of computer and the processing mode. Table A-1 summarizes these commands.

Table A−1 EDT Display Commands

MODE:	SBC, 3B2	3B4000 ACP	3B15, 3B4000 MP	3B4000 EADP
Firmware	**edt**	--	**disp edt**	**disp edt**
DGMON	**show**	--	--	--
From the UNIX command line	**/etc/prtconf** **edittbl** --	-- **edittbl** **getedt**	**/etc/prtconf** -- **getedt**	-- -- **getedt**

The **getedt** and **disp edt** commands are combined into the same subsection, as are the **edt** and **show** commands. The following subsections list the other display commands alphabetically.

edt and show Commands

The 3B2 computer **edt** and the DGMON **show** command are accessed from firmware mode. **show** has exactly the same output as **edt**. NOTE: The 3B4000 ACP does not have a console, so all firmware mode prompts are not usable.

On the 3B2 computers, execute the following commands shown in bold in Figure A-1 after booting:

```
# shutdown -i5 -g0 -y
FIRMWARE MODE

password
Enter name of program to execute [   ]: edt

Current System Configuration

System Board memory size: 12 megabyte(s)
#0 - 4 megabyte(s), #1 - 4 megabyte(s), #2 - 2 megabyte(s), #3 - 2 megabyte(s)

00 - device name = SBD       , occurrence = 0, slot = 00, ID code = 0x01
     type = integral i/o bus
     boot device = y, board width = double, word width = 2 byte(s)
     req Q size = 0x00, comp Q size = 0x00
     subdevice(s)
     #00 = FD5       , ID code = 0x01

Press any key to continue

01 - device name = SCSI      , occurrence = 0, slot = 01, ID code = 0x100
     type = integral i/o bus
     boot device = y, board width = single, word width = 2 byte(s)
     req Q size = 0x38, comp Q size = 0x38, indirect edt
     subdevice(s)
     #00 = disk      , ID code = 0x100, #01 = tape      , ID code = 0x101

Press any key to continue

Enter name of program to execute [ ]: /unix
```

Figure A−1 Testing the EDT on a 3B2 Computer

In Figure A-1, the first command line (**shutdown**) brings the system down to single user mode and then to firmware mode. *password* is the firmware password, usually **mcp**. At the "Enter name..." prompt, **edt** displays the EDT, and **/unix** takes you back to multiuser mode. Refer to the System Administration Guide supplied with your system for more information on bringing a computer to firmware or to the diagnostic monitor modes.

This display is for a 3B2 600 computer, but each 3B2 computer will have a similar display.

getedt and disp edt Commands

On the 3B4000 MP, adjunct processors, or the 3B15 computer, to display the EDT, use the **disp edt** command from firmware mode or the **getedt** (see Table A-2) command when the UNIX system is running.

Table A−2 3B4000/3B15 getedt Listing

System EQUIPPED DEVICE TABLE

BD CODE	DEV SIZE	DEV TYPE	DEVICE NAME + NUMBER		ADDRESS	AUTO CNTL	INT LEV	UNIT EQUIPAGE	PHNUM	ROMSZ	RELS.	DATE
1	4	1	CCS	0	-	0	4bd01ad	19	20000	102	1087	
2	1	2	MASC	0	100000	-	0	ffffff	37	4000	101	483
1	4	1	CCC	0	-	0	4	18	20000	102	1087	
3	4	21	TAPE	0	180000	-	3	0	22	8000	103	485
4	4	89	SLIC	0	200000	-	5	0	311	20000	22	1286
5	2	11	IDFC	0	280000	-	5	10073	32	10000	102	685
6	4	1	ABI	0	300000	-	5	16	200000	1	486	
7	1	2	ADLI	0	380000	-	3	0	11	4000	101	483
8	2	1	MAU	0	400000	-	4	1	21	8000	0	0
9	4	1	IOA	0	480000	-	3	0	16	10000	103	584
a	1	2	SDLI	0	500000	9	3	0	18	4000	101	483
b	1	2	ADLI	1	580000	9	3	0	11	4000	101	483
c	4	1	IOA	1	600000	-	3	0	16	10000	103	584
d	2	11	IDFC	1	680000	-	5	3e373	32	10000	102	685
e	2	1	SADL	0	700000	9	3	0	15	8000	102	685
f	2	1	NI5	0	780000	-	4	1	24	8000	201	185

EXTENDED EQUIPPED DEVICE TABLE FOR SLIC AT ADDRESS 200000

MAJ NUMBER	DEVICE NAME + NUMBER		DEVICE TYPE	EQUIPPED LOGICAL UNITS
4	HA	0	3	NONE
114	DISKTD	1	1	0
120	DISKTD	7	1	0
4	HA	8	3	NONE

This display is from the **getedt** command; the firmware **disp edt** command gives a listing with an 8 to the left of the first column to identify bootable devices. The definitions of these columns are

- **BD CODE** — the board code. For hardware devices (except those on the extended bus), this is the major number. In this configuration, devices from the first ADLI have the major number 3; devices from the second have the major number 10 (indicated by the a). This number corresponds to the board code on the bus. The number is the major number for boards on the primary and growth units. Refer to the Operations and Administration Guide supplied with your system for information on major numbers on

extended buses.

- **DEV SIZE** — the device size; the number of bits used to address a board. "1" indicates 1-byte or 8 bits (every byte is addressable), "2" indicates 2 bytes or 16 bits (that every half word is addressable), "3" indicates 2 bytes or 16 bits (that every other half word is addressable), and "4" indicates 4 bytes or 32 bits (every word is addressable). Number "2" or "3" means that boards can be addressed with 8 or 16 bits; number "4" means that 8, 16, or 32 bits can be used. NOTE: "3" is not implemented at this time.
- **DEV TYPE** — the device type; the type of circuit board. The right digit is 1 for an I/O controller board, 2 for an I/O interface board. The left digit indicates a copy device, where 1 represents a disk copy device and 2 indicates a tape copy device.
- **DEVICE NAME** — the device name designation for this type of circuit board.
- **DEVICE NUMBER** — all circuit boards of the same type are numbered, beginning with 0, in this column to differentiate them. Disk drive 0 must be connected to IDFC 0 for booting purposes.
- **ADDRESS** — device address code reference from the local bus address of the demand paging central controller (DPCC) boards.
- **AUTO CNTL** — automatic controller; the board code of the controlling circuit board. For example, for ADLIs, SDLIs, and SADLs, this is the board code of the IOA by which they are controlled.
- **INT LEV** — the interrupt level at which a circuit board is served by the Central Control and Cache (CCC). The higher the number, the greater the interrupt priority.
- **UNIT EQUIPAGE** — device dependent equipment data base.
- **PHNUM** — phase number; the total number of diagnostic phases for this device. Refer to Appendix B for more information on diagnostic phases.
- **ROMSZ** — the amount of on-board read-only memory (ROM), expressed in bytes.
- **RELS and DATE** — the release version of the board and the date (month and year) the firmware was released.

The definitions of the columns in the extended EDT for SCSI are

- **MAJ NUMBER** — The major external device number for the SCSI device.
- **DEVICE NAME** — The name of the device. These names are administered by and registered with AT&T.
- **DEVICE NUMBER** — All circuit boards of the same type are numbered, beginning with 0, in this column to differentiate them.

- **DEVICE TYPE** — The SCSI subdevice supported by the specified device. In the **getedt** listing, DISKTD is the SCSI disk drive, HA is the SCSI Host Adapter that allows the device-independent SCSI bus to communicate with the device-dependent host computer.
- **EQUIPPED LOGICAL UNITS** — The logical disk or tape (logical unit) number. This number is either 0 or NONE. SCSI target controllers on the 3B4000 computer support one device, labeled 0. NONE indicates that no devices are supported.

The **getedt** EDT listing for the SCSI devices on the 3B4000 ACP is shown in Table A-3.

Table A−3 3B4000 ACP getedt Listing

```
System EQUIPPED DEVICE TABLE FOR PE=# TYPE=ACP MEMORY=NNNNNNNN

 OPT    WORD    OPT          DEVICE          DEV    SMRT    DIAG
 CODE   SIZE    TYPE       NAME + NUMBER     SLOT   BRD     FILE
  1      1       0          SBD      0        0      1      SBD
 100     1       0          SCSI     0        1      1      SCSI



EXTENDED EQUIPPED DEVICE TABLE FOR SCSI AT slot 1

 MAJ          DEVICE          DEVICE      EQUIPPED
NUMBER     NAME + NUMBER       TYPE     LOGICAL UNITS

 121       SD01      1          1      0 1
```

The definitions of these columns are

- **OPT CODE** — Same as the ID_code[1] in the firmware EDT display, a number between 0x0 and 0xffff that a device uses to identify itself. ID codes must be registered with and are administered by AT&T. Some devices are assigned special opt codes. Coprocessors are assigned numbers starting at 0xfd00; unbuffered microbus devices are assigned numbers starting at 0xfe00; and buffered microbus devices are assigned numbers starting at 0xff00.
- **WORD SIZE** — The word size of a device I/O bus. A "1" indicates devices with a bus word greater than 8-bits; a "0" indicates devices with an 8-bit bus word.

1. ID_code appears in a listing created with the **edittbl**(1M) command. This command is described later in this chapter.

- **OPT TYPE** — The type of I/O bus (seen Table A-4) associated with the device.

Table A−4 I/O Bus Types

Value	Bus Type
0	Integral I/O Bus Slot
1	Coprocessor Slot
2	Unbuffered Microbus Slot
3	BUffered Microbus BUS (BUBUS) Slot
7	Miscellaneous Slot

- **DEVICE NAME** — Field name for a device. Device names are administered by AT&T. This string is also the field name that DGMON loads to diagnose a device.
- **DEVICE NUMBER** — All circuit boards of the same type are numbered, beginning with 0, in this column to differentiate them. Disk drive 0 must be connected to IDFC 0 for booting purposes.
- **DEV SLOT** — The device slot is the physical slot number in which the board resides.
- **SMRT BRD** — The smart board designation indicates whether the device is intelligent, meaning either that it requires downloaded code for normal operation or supports subdevices. A "1" indicates an intelligent device; a "0" specifies a "dumb" device.
- **DIAG FILE** — The name of the diagnostics file in the */adj/pe#/dgn* directory.

The definitions of the columns in the extended EDT for SCSI are

- **MAJ NUMBER** — The major external device number for the SCSI device.
- **DEVICE NAME** — The name of the device. These names are administered by and registered with AT&T.
- **DEVICE NUMBER** — All circuit boards of the same type are numbered, beginning with 0, in this column to differentiate them.
- **DEVICE TYPE** — The SCSI subdevice supported by the specified device. In the **getedt** listing, SD01 is the SCSI disk drive.
- **EQUIPPED LOGICAL UNITS** — The logical disk or tape (logical unit) number. This number is either 0, 1, or NONE. SCSI target controllers on the 3B4000 ACP supports up to two devices with 0 indicating the floppy disk driver, and the one indicating a hard disk driver. NONE indicates that no devices are supported.

/etc/prtconf Command

To display the EDT, use the following UNIX system command

/etc/prtconf

A sample display from **/etc/prtconf** is shown in Figure A-2.

```
AT&T 3B2 SYSTEM CONFIGURATION:

Memory size: 2 Megabytes
System Peripherals:

   Device Name   Subdevices          Extended Subdevices

   SBD
                 Floppy Disk
                 72 Megabyte Disk
   SCSI
                 SD01 ID1
                                     147 Megabyte Disk ID0
                 ST01 ID2
                                     TAPE ID0
   PORTS
   MAU
```

Figure A−2 Sample /etc/prtconf Display

The definitions for the columns are

- **Device Name** — a name taken from the *edt_data* file when the computer is booted.
- **Subdevices** — the names of subdevices associated with the device. These names are built into the **/etc/prtconf** program. When additional devices are added to the *edt_data*, and **prtconf** cannot obtain all of the information for the device, a new **prtconf** program must be created and placed in the */etc/prtconf.d* directory.

Field Comparisons of EDTs for Different Systems

The following table (Table A-5) shows which fields correspond for the EDTs on the different systems. This information is useful when you are examining multiple EDTs.

Table A–5 EDT Fields By System

3B2	3B15/3B4000 MP	3B4000 ACP
ID_code (hexadecimal)	-- --	*OPT Code (hexadecimal)*
-- --	*Board Code (hexadecimal)*	*Major Number (Extended EDT Table) (decimal)*
dev_name	*Device Name*	*dev_name*
rq_size	--	*rq_size*
cq_size	--	*cq_size*
boot_dev	*[embedded in Board Code]*	*boot_dev*
word_size 1=16-bit 0=8-bit	*Device Size 1=8-bit 2,3=16-bit 4=32-bit*	*word_size 1=16-bit 0=8-bit*
brd_size	--	*brd_size*
smrt_brd	--	*smrt_brd*
cons_cap	--	--
cons_file	--	--
indir_dev	--	*indir_dev*
--	*Device Type*	*Device Type (Extended EDT Table)*
--	*Device Number*	*Device Number*
--	*Device Address*	--
--	*Auto Control*	--
--	*Interrupt Level*	--
--	*Unit Equipage*	--
--	*Phase Number*	--
--	*ROM Size*	--
--	*Release and Date*	--
--	--	*OPT Type*
--	--	*Device Slot*
--	--	*Diagnostics File*

/dgn/edt_data, The EDT Initialization File

On the SBC, 3B2 computer, and 3B4000 ACP, the */dgn/edt_data* file lists all hardware devices that may be configured on the system. The **filledt**(8) process uses this file to search for hardware devices, and adds any that are found to the EDT (only when the system is booted). The *edt_data* file is supplied with a computer when purchased and is upgraded automatically when AT&T add-on products are installed. Your installation package should do this task as well. When installing a driver for the first time with a new piece of hardware, use **edittbl** with the -i option to add the appropriate information to *edt_data*. The command syntax is

```
/etc/edittbl /dgn/edt_data -d -i
```

To display the *edt_data* table, use the following command:

```
/etc/edittbl /dgn/edt_data -l -d
```

SBC edt_data File

The **/etc/edittbl** display for the SBC is shown in Figure A-3.

```
num_dev: 0x2

ID_code: 0x0001 dev_name:   SBD          dev_addr: f8000000

ID_code: 0x0003             dev_name:    PORTS
```

Figure A−3 SBC /etc/edittbl Display

The definitions of these fields are

- **num_dev:** The number of devices described in the listing.
- **ID_code:** A number between 0x0 and 0xffff that a device uses to identify itself.
- **dev_name:** Field name for a device. This string is also the field name that DGMON loads to diagnose a device.
- **dev_addr:** Physical address that can be read (single-byte read) to detect the device.

3B2 edt_data File

The **/etc/edittbl** display for the PORTS and EPORTS boards on the 3B2 computer is shown in Figure A-4 (from a 3B2 500 computer).

```
ID_code: 0x0003   dev_name:     PORTS   rq_size: 0x03   cq_size: 0x23
boot_dev: 0       word_size: 1          brd_size: 0     smrt_brd: 1
indir_dev: 0      cons_file: 1

ID_code: 0x0102   dev_name:    EPORTS   rq_size: 0x21   cq_size: 0x46
boot_dev: 0       word_size: 1          brd_size: 0     smrt_brd: 1
indir_dev: 0      cons_file: 1
```

Figure A-4 3B2 Computer /etc/edittbl Display

The definitions of these fields are

- **ID_code:** A number between 0x0 and 0xffff that a device uses to identify itself. ID codes must be registered with and are administered by AT&T.
- **dev_name:** Device name; a field name for a device. Device names are administered by AT&T. This string is also the field name that DGMON loads to diagnose a device.
- **rq_size:** Request queue size; a number between 0x0 and 0xff that represents the count of entries in a device's job request queue.

- **cq_size:** Completion queue size; a number between 0x0 and 0xff that represents the count of entries in a device's job completion queue.
- **boot_dev:** Boot device; indicates whether this device can be used to boot the system. A "1" means that it is bootable; a "0" means that it is not.
- **word_size:** The word size of a device I/O bus. A "1" indicates devices with a 16-bit bus word; a "0" indicates devices with an 8-bit bus word.
- **brd_size:** Board size; specifies the I/O connector slots that a device requires. A "1" indicates that two slots are needed; a "0" indicates that one slot is required.
- **smrt_brd:** Smart board; indicates whether the device is intelligent, meaning either that it requires downloaded code for normal operation or supports subdevices. A "1" indicates an intelligent device; a "0" specifies a "dumb" device.
- **cons_cap:** Console capability; shows whether this device can support the system console terminal. A "1" is used for devices that can; a "0" for those that cannot.
- **indir_dev:** Indirect device; indicates whether all the information on the subdevices associated with a device can be directly accessed by **/etc/prtconf**. Indicate "0" if all the information for a device is directly accessible. Indicate "1" if subdevice information must be determined by another program. If "1" is indicated, a special file for getting information about the subdevices must reside in the */etc/prtconf.d* directory. Refer to the end of this appendix for an example of the *prtconf.c* file.
- **cons_file:** Console file; indicates whether a device that can support the system console terminal requires extra code to do so. This feature is not supported and the value in this field is not evaluated.

To display the EDT for a subdevice, use the command

/etc/edittbl /dgn/edt_data -l -s

SBC Subdevice Display

The subdevice display generated for the SBC is shown in Figure A-5.

```
num_sbdev: 0x1

Device: XXXX     (0x000a)   Unit: 0   subdev_name: Hard
```

Figure A–5 SBC Subdevice Display

The definitions of these fields are

- **Device:** Field name for a device. This string is also the field name that DGMON loads to diagnose a device.
- **(0x*number*):** The identification code (ID_code). A number between 0x0 and 0xffff that a device uses to identify itself.
- **Unit:** The subdevice number. This information conforms to the maximum number of subdevices per device defined in the #DEV column of the */etc/master.d* file for the driver.
- **subdev_name:** The name assigned to the subdevice (a designation for a type of device). Subdevice names are all uppercase and one to nine characters long. Can be either the device type (Hard, Floppy, cartridge, Serial, Bootable) or the actual board name (HD20, FD5, and so on).

3B2 Computer Subdevice Display

The 3B2 computer subdevice display is shown in Figure A-6.

```
num_sbdev: 0xe
ID_code: 0x0000 subdev_name: NULL
ID_code: 0x0001 subdev_name: FD5
ID_code: 0x0002 subdev_name: HD20
ID_code: 0x0003 subdev_name: HD30
ID_code: 0x0005 subdev_name: HD72
ID_code: 0x0006 subdev_name: HD72A
ID_code: 0x0007 subdev_name: HD72B
ID_code: 0x0008 subdev_name: HD72C
ID_code: 0x0009 subdev_name: HD43
ID_code: 0x000a subdev_name: HD72D
ID_code: 0x0100 subdev_name: disk
ID_code: 0x0101 subdev_name: tape
ID_code: 0x0104 subdev_name: worm
ID_code: 0x0004 subdev_name: FT25
```

Figure A-6 3B2 Computer Subdevice Display

The definitions of these fields are

- **num_sbdev:** Indicates how many subdevices are associated with the device.
- **ID_code:** Number that identifies a subdevice, in the range 0x0 to 0xfff. Subdevice ID codes are administered by and must be registered with AT&T.
- **subdev_name:** Designation for this type of device. Subdevice names are all uppercase, one to nine characters, and are administered by and must be registered with AT&T.

Adding Entries to a 3B15/3B4000 Master Processor EDT

On the 3B15 and 3B4000 computers, any properly-installed board will be added to the EDT at boot time. This requires the following:

- The ID register must be hard-assigned in the firmware of the board.
- The On-board Device Information Table (ODIT) structure must be hard-assigned in the firmware at 0x48F. The ODIT contains the board's generic name, release and point issue, and the date stamp from inside the PROMs. The structure of the ODIT is defined in the *firmware.h* file.
- Three bergs (connectors) must be installed on the pins of the backplane. These assign the local bus address, the interrupt level, and the bus arbitration level for the board (already present and must be adjusted).
- The board must be properly installed in the slot.

To check the hardware installation, check **disp edt** in firmware mode to validate the fields, and then boot the system with the hardware in place but without a master or */boot* file for the device. If the hardware is correctly installed, you will get a message that the driver was not found.

Adding Devices to the SBC, 3B2 Computers, and the 3B4000 ACP EDT

If you are installing a new piece of hardware not supplied as an AT&T add-on, you must manually add entries for new the device to the */dgn/edt_data* table that is used to create the EDT. Note that none of the changes you make in the *edt_data* file actually affect the configuration of the computer until it is rebooted. If you make a mistake, remove the entry (refer to that section for more details), and insert it again until correct. When you are using **edittbl** on a 3B4000 ACP, include the **-P** option to specify the proper processing element. The steps for inserting an entry in the EDT are

1 In the */dgn* directory (or */adj/pe#/dgn* on an adjunct), make a copy of the *edt_data* file that you can use to recover from a mistake

```
cp edt_data hold.edt_data
```

2 View the existing contents of the EDT

```
edittbl -l -d -s
```

3 Ensure that the *edt_data* has write permission enabled.

4 Add information about the new device

```
edittbl -d -i
```

5 Add information about subdevices for the new device. Note that every device must have at least one subdevice or it will be ignored. If necessary, you can use the subdevice name "Other" to create a phantom subdevice.

Exit by typing **q** or CTRL-d to the **device ID** prompt.

6 Verify your entry in *edt_data*

```
edittbl -l -d -s
```

7 When you are finished, reboot your system so that the new EDT is recognized.

8 Verify that the device was included in the EDT by running the **/etc/prtconf** command.

EDT Command Examples

In the following examples, the computer prompts are in `constant width` type, the programmer responses are in **bold** type. The computer does not update the file until after all the information is entered; if you quit inserting by entering **q** or by pressing BREAK or DELETE , the file is not changed. Enter "." or CTRL-d to complete entering data. No validity checking is done on the

information you enter, if a value does not correspond to the device, the boot software will not be able to load the device and will fail. If you enter a value that is out of range, for example, specifying a completion queue size of 0xffff, **edittbl** will truncate the value down to the maximum value, 0xff.

If you enter data that you later discover is incorrect, you can remove the entry by using **edittbl** with the **-r** option. The prompts for this option are the same as for the **-i** option. All of the information for the entry being removed must match that entered originally for the entry.

Adding an Entry to the EDT on an SBC

Figure A-7 is a session to add the fictional XXXX device to the EDT.

```
# edittbl -d -l
utility program for edt_data

num_dev: 0x1

ID_code: 0x0000 dev_name:          SBD       dev_addr:0
# edittbl -d -i
utility program for edt_data

Enter device data

Enter device ID code: 0x1
Enter device name: XXXX
Device address? : 0xffff8000

Enter device ID code: 0x.
```

Figure A−7 Adding an Entry to the SBC EDT Example

You should enter the following information for each prompt:

1. **Device ID code:** Use the next available number. This number is used only to associate a subdevice with a device and does not correspond to other numbers
2. **Device name:** Use the same name as the file in */boot* in all uppercase letters.
3. **Device address:** Physical address that can be read to detect the device. At system boot time, **filledt**(1M) reads a byte at the device address. If something responds to the read, the device is considered present and is logged into the EDT.

Figure A-8 shows how a subdevice is added to the EDT for the SBC.

```
# edittbl -s -i
utility program for edt_data

Enter device data

Enter device ID code: 0x1
Enter subdevice unit: 0
Enter subdevice name: Hard

Enter device ID code: 0x.
#
```

Figure A−8 Adding an SBC Subdevice Example

You should enter the following information for each prompt:

1. **Device ID code:** Use the same number that was specified when the device was added to the EDT.
2. **Subdevice unit:** Start at 0 and increase sequentially. Ensure that this information conforms to the maximum number of subdevices per device defined in the #DEV column of the */etc/master.d* file for the driver.
3. **Subdevice name:** Designation for this type of device. Subdevice names can be upper or lowercase and are one to nine characters long. Can be either the device type (Hard, Floppy, cartridge, Serial, Bootable) or the actual board name (HD20, FD5, and so on).

Adding an Entry to the EDT on a 3B2 Computer

The following is a session to add the fictional THUD device to the EDT. Information in *[italics]* provides a reference to the names displayed when **edittbl** is used to list the *edt_data* file. Refer to the previous section on displaying the EDT on a 3B2 computer for more information about individual prompts.

```
# edittbl -d -l
utility program for edt_data

num_dev: 0x2

ID_code: 0x0001 dev_name:        SBD     rq_size: 0x00   cq_size: 0x00
boot_dev: 1     word_size: 1     brd_size: 1     smrt_brd: 1     cons_cap: 1
indir_dev: 0    cons_file: 0

ID_code: 0x0003 dev_name:      PORTS     rq_size: 0x03   cq_size: 0x23
boot_dev: 0     word_size: 1     brd_size: 0     smrt_brd: 1     cons_cap: 1
indir_dev: 0    cons_file: 1

# edittbl -d -i
utility program for edt_data

Enter device data

Enter device ID code (> 0x10000 if indirect): 0x5   [ID_code]
Enter device name: THUD                            [dev_name]
Enter request queue size: 0x0                      [rq_size]
Enter completion queue size: 0x0                   [cq_size]
Boot device? (1 - yes / 0 - no): 0                 [boot_dev]
16 bit I/O bus? (1 - yes / 0 - no): 0              [word_size]
Double width board? (1 - yes / 0 - no): 0          [brd_size]
Intelligent board? (1 - yes / 0 - no): 1           [smrt_brd]
Console Capability? (1 - yes / 0 - no): 1          [cons_cap]
Console pump file? (1 - yes / 0 - no): 0           [cons_file]


Enter device ID code (> 0x10000 if indirect): 0x.
```

Figure A–9 Adding a 3B2 Device Example *(part 1 of 2)*

```
# edittbl -l -d
utility program for edt_data

num_dev: 0x3

ID_code: 0x0001 dev_name:        SBD     rq_size: 0x00   cq_size: 0x00
boot_dev: 1     word_size: 1     brd_size: 1     smrt_brd: 1     cons_cap: 1
indir_dev: 0    cons_file: 0

ID_code: 0x0003 dev_name:      PORTS     rq_size: 0x03   cq_size: 0x23
boot_dev: 0     word_size: 1     brd_size: 0     smrt_brd: 1     cons_cap: 1
indir_dev: 0    cons_file: 1

ID_code: 0x0005 dev_name:       THUD     rq_size: 0x00   cq_size: 0x00
boot_dev: 0     word_size: 0     brd_size: 0     smrt_brd: 1     cons_cap: 1
indir_dev: 0    cons_file: 0

# edittbl -s -i
utility program for edt_data

Enter subdevice data

Enter subdevice ID code: 0x34
Enter subdevice unit: 0
Enter subdevice name: Hard

Enter subdevice ID code: 0x.
```

Figure A−9 Adding a 3B2 Device Example *(part 2 of 2)*

Removing an Entry From the EDT

The **edittbl** command contains the **-r** option for removing an entry from the EDT. This option prompts you for information and then uses that information to remove the appropriate device from the *edt_data* file. NOTE: Removing an entry has no effect until the system is rebooted. When you execute **edittbl -r**, the command prompts you for the same information you specified for inserting an entry. However, only the ID_code is used to detect the entry to be removed from *edt_data*.

When a device is removed from the EDT, all associated subdevices are also removed. As with inserting an entry, use "." or CTRL-d to end the data input.

Appendix B: Writing 3B2 Computer Diagnostics Files

Contents

Appendix B: Writing 3B2 Computer Diagnostics Files

This appendix explains how to write diagnostics files. The two diagnostic files are referred in this appendix as a *diagnostics design*. The Appendix B shows a complete diagnostics design for a custom feature card (non-common I/O based card) including examples for all required files. The code examples listed in this appendix can be used as a template for writing diagnostics.

Common I/O is a specification for circuit board design that ensures that bus-to-processor communication is standardized. The design specified in this appendix does not utilize common I/O.

The first part of this appendix serves as background information for the organization of diagnostic files for the 3B2 computer family. The second part describes the diagnostic programs or modules that are necessary for proper operation.

A diagnostic file passes information to an intelligent controller so that the system initialization software can ensure the integrity of a 3B2 computer feature card (circuit board). Each hardware device requires two diagnostics files and these files are stored in the */dgn* directory. Both file names are in uppercase and both have the same name as the driver's master file name, except that one file is prefaced with *X*. The *X.* file contains object code to be downloaded to the feature card. The other file is an object file, which is to be loaded into main memory and executed by the CPU. Figure B-1 illustrates these two files for the **mydev** device.

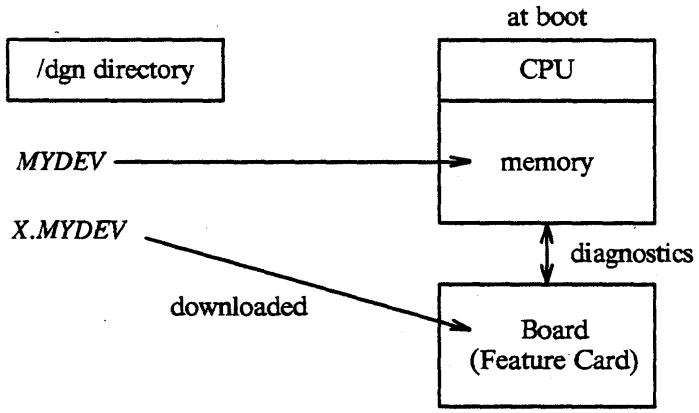


Figure B–1 Diagnostics Files Overview

If downloading is unnecessary, a NULL object file must be supplied such as *SBD* and *X.SBD*. Link the name of your product to */dgn/SBD* and *X.product-name* to *X.SBD*. For example, for the **nodev** device

```
cd /dgn
ln SBD NODEV
ln X.SBD X.NODEV
```

Introduction to Diagnostics Programs

The diagnostic programs on a 3B2 computer are part of the Maintenance and Control Program (MCP). The MCP has two operation modes

- **Noninteractive mode** — a mode in which the integrity of a 3B2 computer hardware is checked automatically when the computer is powered up, or at any time that the computer is brought down to firmware mode and back to a multiuser mode. Because a 3B2 computer can bring itself up into full multiuser mode without user intervention, the noninteractive mode of the MCP is also referred to as autoboot mode.

- **Interactive mode** — a mode of the MCP in which the integrity of a 3B2 computer hardware is checked when specifically requested. This mode is entered from either multiuser mode or automatically when hardware or system software failures occurs. In interactive mode, more extensive diagnostics can be run.

MCP Noninteractive Mode

Noninteractive (autoboot) mode is entered when the computer is powered on. A total system reset occurs at this time and basic sanity checks are performed on the computer hardware. The sanity checks include testing the processor (CPU), the Memory Management Unit (MMU), the erasable programmable read-only memory (EPROM), the non-volatile random-access memory (NVRAM), the Integral Dual Universal Asynchronous Receiver-Transmitter (IDUART), and the first 16 kilobytes of dual-ported dynamic RAM.

If a problem occurs during the sanity checks, the front panel diagnostic indicator light emitting diode (LED) pulses on and off in a defined pattern to identify the type of sanity failure.

Table B-1 defines the LED patterns.

Table B−1 Diagnostic Indicator LED Patterns

Pulse Count	Failure Type
1	System is in a firmware null state with no console device; connect a terminal to the default console port
2	Processor sanity test failed
3	(EP)ROM sanity test failed
4	RAM (first 16k) sanity test failed
5	IDUART sanity test failed

After the sanity checks are done, a self-configuration process takes place by the MCP calling **filledt**(8) to identify and locate all of cards on the bus. (**filledt** resides in the root directory.)

As self-configuration terminates, a more extensive diagnostic run begins. All diagnostics for the 3B2 computer are under the control of **dgmon**(8), the diagnostic monitor. The **dgmon** program resides in the root directory and is invoked by noninteractive MCP. **dgmon** loads the diagnostic files from the */dgn* directory of the integral hard disk into main memory and executes them.

MCP Interactive Mode

The MCP interactive mode is entered only when a failure condition occurs for disk diagnostics, self-configuration, boot or by means of a specific request of the UNIX operating system. Entry to the MCP interactive mode is also possible by activating the reset button during a diagnostic sequence, which simulates a failure condition.

The procedure to enter the MCP interactive mode is

1 Bring the computer to *init 5* state with the **shutdown**(1M) command

shutdown -i5 -g0 -y<CR>

2 Upon entering the MCP interactive mode, the console displays

```
FIRMWARE MODE
```

If entry to interactive MCP is made from any of the failing conditions previously described, the console displays

```
SYSTEM FAILURE: CALL YOUR SERVICE REPRESENTATIVE
```

Accessing the MCP

All 3B2 computers are factory equipped with the Maintenance and Control Program (MCP) password **mcp**. (This default password can be changed using the interactive MCP **passwd**(8) command.) The MCP is accessed as follows

1 At the prompt, enter the password. The entry is not displayed on the console.

2 After the password is entered, the console displays one of the following messages

```
Enter Name of Program To Execute [ ]:
```

or

```
3B2 Monitor/Control Program - erase 'H', kill '@'
Physical Mode
```

To enter the MCP interactive mode on machines equipped with DEbug MONitor (DEMON) EPROMS, enter

```
> boot
```

The system responds

```
Enter Name of Program To Execute [ ]:
```

When the computer is in the interactive mode of MCP, the following firmware-resident programs (see Table B-2) can be executed

Table B−2 Interactive MCP Commands

Program	Description
baud	change console baud rate
boot	execute a system or user supplied program
edt	display the Equipped Device Table (EDT)
errinfo	display contents of internal registers
express	change automatic diagnostics toggle
newkey	write disk key for NVRAM
passwd	change the firmware password
q or quit	escape to FIRMWARE MODE prompt
sysdump	call **crash**(1M)
version	display firmware version and load data
?	list help information

Each program is described in Section 8 of the *System Administrator's Reference Manual*. Refer to the 1/87 update of the manual for information on **errorinfo** and **express**.

In addition to the firmware resident programs listed, it is possible to execute any user-supplied or system-supplied program resident on one of the available disk storage devices. Two restrictions apply

- The storage device must be present in EDT. A storage device cannot be mounted from firmware mode and, consequently, programs can be retrieved only from the devices that are in the EDT.
- The user program must be loaded above the highest memory location used by the system; location 0x200400 is recommended. When the **boot** command is entered, the MCP asks for the name of the program to execute. The user program does not have to reside in a root directory of the particular storage device. The MCP accepts a fully qualified path name of the file as well.

The boot firmware is also used by the MCP to bring up the diagnostic monitor when a computer is powered on and by the operating system after diagnostics. The difference between the two programs and the user programs is that the fully qualified path is automatically provided by MCP.

Figure B-2 shows the power-up diagnostic sequence for the 3B2 computer.

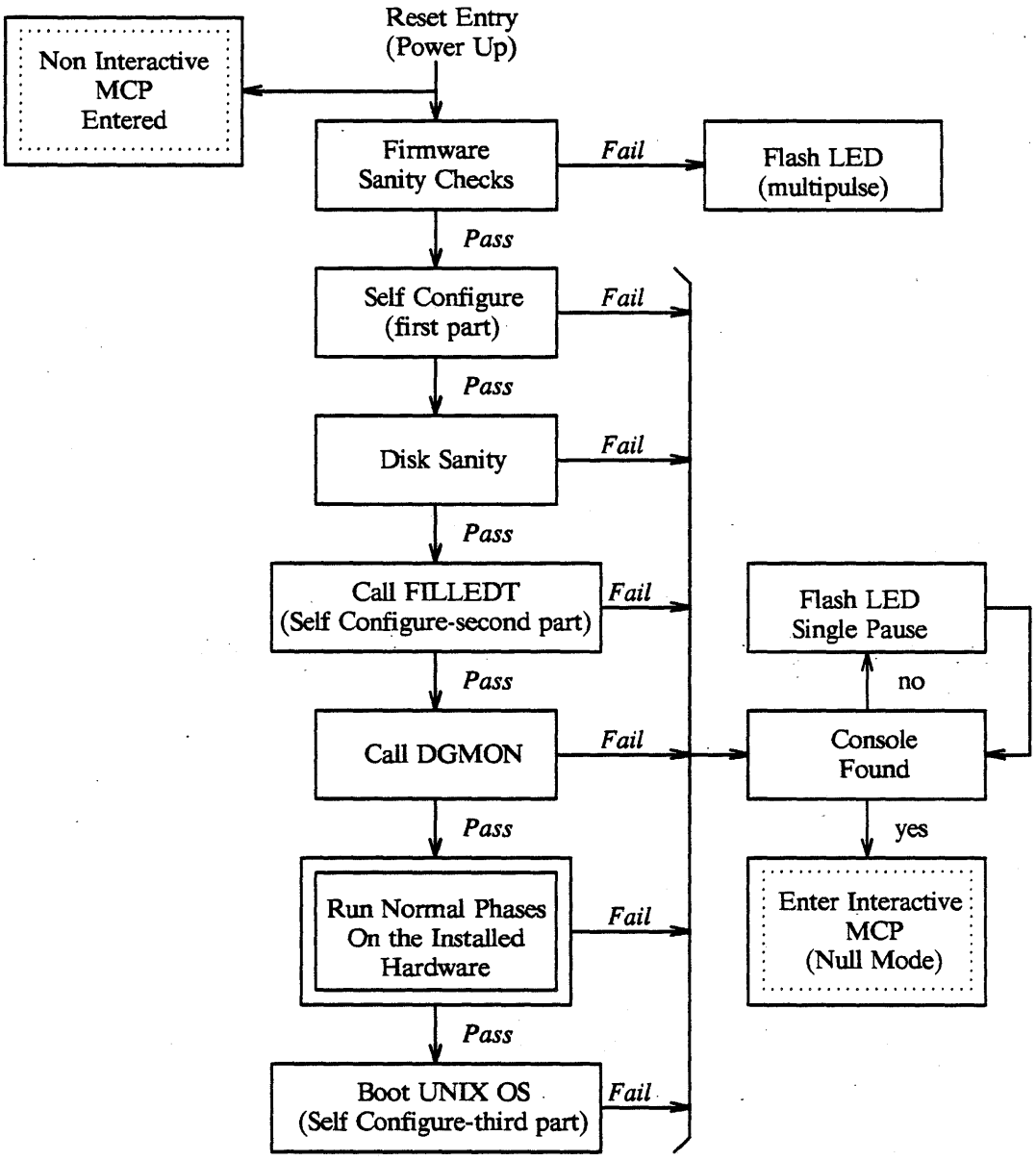


Figure B–2 3B2 Diagnostic Sequence

The Diagnostic Monitor (dgmon)

All diagnostics for the 3B2 computer are under the control of the diagnostic monitor **dgmon**(8). Diagnostics are run during system initialization and are loaded from the integral hard disk. The program **dgmon** also resides in the root (/) directory and is invoked by the noninteractive mode of the MCP. The **dgmon** program loads diagnostics from the integral hard disk's */dgn* directory into main memory and executes the diagnostics.

Diagnostics invoked from interactive MCP mode can be called explicitly and loaded from the integral hard disk, external hard disk (such as a Small Computer System Interface (SCSI)), integral floppy, or other device.

The MCP autoboot mode is used during power up to run normal diagnostics on each peripheral device, including System Board Diagnostics (SBD). Secondly, the demand mode is initiated from the console while in firmware mode.

Typically, you should write several diagnostic programs to test the integrity of custom hardware. These diagnostic programs are called *diagnostic phases*. Any diagnostic program or phase on any peripheral can be run in demand mode. Also, demand mode is the only mode in which interactive phases can execute.

The diagnostic monitor (**dgmon**) can execute a diagnostic program or phase written to test a custom-designed feature card automatically. Because the diagnostic phases are being executed by **dgmon**, the phases must adhere to several rules imposed by **dgmon**. This is necessary to ensure that the results of the test can be interpreted properly and that the syntax for invoking the diagnostic tests through the **dgn** command is uniform for all 3B2 computer peripherals.

The 3B2 computer diagnostics reside in two separate files and are downloaded into main memory from either the hard disk or the floppy disk. One of the diagnostics files contains system board code (**m32** executable) and the other file contains the object code of the processor. A 3B computer peripheral receives (is pumped) the object code that is then executed.

The diagnostic phases shown in this appendix are actual working diagnostics written for the general-purpose 3B2 computer interface card model HR1.

Diagnostic Monitor Commands

The diagnostic monitor is entered from the interactive MCP at the following prompt

```
Enter Name of Program to Execute [ ]:
```

Enter /**dgmon** and press the RETURN key. **dgmon** then displays the following prompt

```
Load Device Option Number [ default loader ]:
```

If your system is equipped with a SCSI bus, the *default loader* message reads `1 (SCSI)`. Press the RETURN key and the following additional prompt is displayed for selecting a SCSI subdevice

```
Enter Subdevice Option Number [0 (disk)]:
```

Again, press the RETURN key. The following diagnostic monitor prompt is displayed

```
        DIAGNOSTIC MONITOR
DGMON >
```

Table B-3 lists the available **dgmon**(8) commands

Table B–3 dgmon Commands

Command	Abbreviation	Description
dgn*	---	diagnose one or more devices
errorinfo	---	enable/disable error info
help	h	list commands and arguments
list	l	list phases for the specified device
quit	q	return to the MCP interactive prompt
run	r	run diagnostic phases
show	s	show equipped device table

*Refer to the *System Administrator's Reference Manual* on the **dgmon**(8) manual page for more information on the **dgn** command and all its options.

Figure B-3 illustrates the diagnostic utility directories in the root file system

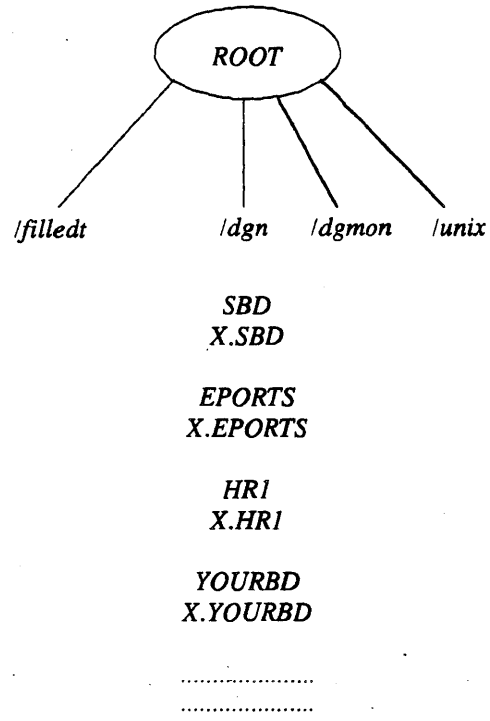


Figure B−3 Diagnostic Utility Directories

Standard Library Functions

A set of common functions, called the *standard library functions*, are available to the diagnostics developer. The standard library functions are a set of macros defined in *firmware.h* that contain calls to the system board code. The functions give diagnostics programs access to custom hardware.

The following is a partial list of the standard library functions. (The HR1 feature card diagnostic phase functions that are used do not appear in the list.) Use both the functions listed here and the HR1 functions when creating the HR1 diagnostic phases.

NOTE: The functions summarized in this appendix (and presented in detail in Section D8X of the *BCI Driver Reference Manual*) should not be confused with similarly named functions in either Section 2 of the *Programmer's Reference Manual* or in Section D3X of the *BCI Driver Reference Manual*. All function names in Section D8X are in uppercase.

Table B-4 summarizes a subset of the standard library functions.

Table B–4 Standard Library Function Subset Summary

Function	Description
EXCRET()	set up return point for exception
EDTP()	
GETS(*ptr*)	get string from standard input
GETSTAT()	return value of current console character
PRINTF("*string %options*",*arg1*,*arg2*)	display message
SSCANF(*string*,"*%options*",*arg1*,*arg2*)	read from *string*
STRCMP(*string1*,*string2*)	compare strings

Writing Diagnostic Phases

Maintenance is an important part of the AT&T 3B2 computer. The maintenance for a 3B2 computer is comprised of diagnostic programs as well as hardware replacement or repair. In addition to the hardware diagnostics (for example, the system board, memory, disk drives, and so on), diagnostics are also run on option feature cards installed in a 3B2 expansion bus. All of the above is done to ensure hardware integrity. If for any reason there is a problem in the system, the console operator should be alerted.

The same is true for a custom feature card development computer. The 3B2 computer with the appropriate diagnostic files is used as a sophisticated test setup to ensure proper operation of the feature cards before the cards are sent to a customer.

Typically, the diagnostics run are more extensively than diagnostics used when the machine is first autobooted. Normal diagnostics, called noninteractive phases, are run automatically when the system is powered up and more extensive diagnostics are run upon demand (called interactive or demand phases).

Diagnostic Files

Every option feature card has to have two files on the 3B2 computer hard disk in the */dgn* directory for diagnostics to be activated. The two files contain instructions that direct the diagnostic monitor **dgmon** to test a specified hardware unit.

dgmon provides information to each phase to indicate the position of a hardware device in the EDT. The diagnostic phase interface consists of a structure containing all necessary information pertaining to the phase target. When the address of a feature card (slot number) or a type of feature card changes, the phase should not be affected because its only interface to the feature cards and the computer resident hardware is not direct but through the **dgmon**.

If the two diagnostic files do not exist in the */dgn* directory, then the diagnostics fail. The computer must pass the diagnostic tests so it may progress to multiuser mode.

System Board Resident Diagnostic Files

The first diagnostic file has the same name as the name of the feature card it serves. It is declared in the EDT. Refer to **edittbl**(1M) in *System Administration Reference Manual* for more information.

For example, if the name of the feature card in the EDT is HR1, then the name of the system board based diagnostic file in the */dgn* directory is HR1. This file contains the system board resident code for diagnostic phases with accompanying phase table.

The system board resident diagnostic file is the only file required to exercise a feature card that cannot download programs. Typically, such a feature card has an onboard microprocessor that executes its program from ROM memory rather then from the downloadable RAM. The system board resident diagnostic program interacts with the microprocessor on the feature card to assign jobs to be performed and to collect data from the feature card.

NOTE: The system board resident diagnostic file **must** be loaded into main memory at address 0x200c000 (hexadecimal). This address is stored in the DOWNADDR constant defined in *diagnostics.h*. After the system board diagnostic file is loaded by **dgmon**, **dgmon** begins execution of every diagnostic phase at this address. Other diagnostic files can be loaded anywhere after this address.

The system board diagnostic file downloads the executable file into feature card memory and executes it there. The next section describes the feature card object code. This file type is in the *m32.out* file format.

Figure B-4 shows the utilization of system board diagnostic RAM for ROM-based feature card diagnostics.

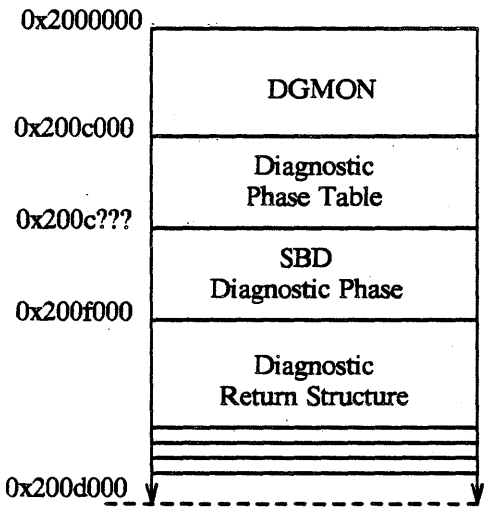


Figure B–4 Utilization of System Board Diagnostic RAM for the HR1 Card

Feature Card Resident Diagnostic File

The second diagnostic file (in the *dgn* directory) for the hardware device is the file containing the feature card object code for the diagnostic tests. Its name is formed by prefacing the file name with *X.* to the system board resident code file. For example, HR1 converts to *X.HR1*. This file is optional, and cannot be zero bytes in length.

Feature cards that can download programs into local card memory, use the *X.* file. The *X.* file can be either in common object format or contain data that is used to create the device object code in memory of the feature card. In either case, the *X.* file is object code that is usable by a processor on the feature card.

For a common I/O feature card, this file is **x86** executable format common object code. This type of object code is compiled and loaded in accordance with *ifile* specifications. File section headers are created to specify the location for the disk to download to the system board memory.

When the *X.* file is not in common object code format (such as when the feature card is not a common I/O feature card or when a 3B2 computer compiler does not exist for a given processor), **dgmon** attempts to read the file into memory as raw data, starting at the END phase address. If the feature card can download programs, you can download from a 3B2 computer hard disk.

Refer to Figure B-5 for a description of system board memory on feature cards that can download programs.

Diagnostic Return Structure

A section of main memory starting at the location 0x200f000 has been allocated as the communication channel between phases. The structure defined for this purpose consists of four unsigned integers starting at location 0x200f000. If this address and structure is not satisfactory for your needs, you may create your own structure or define your own memory address. However, this address and structure are recommended and should be used whenever possible to avoid contention problems at other addresses.

Figure B-5 illustrates how system board diagnostic RAM is used for feature card diagnostics when feature cards are downloaded (pumped).

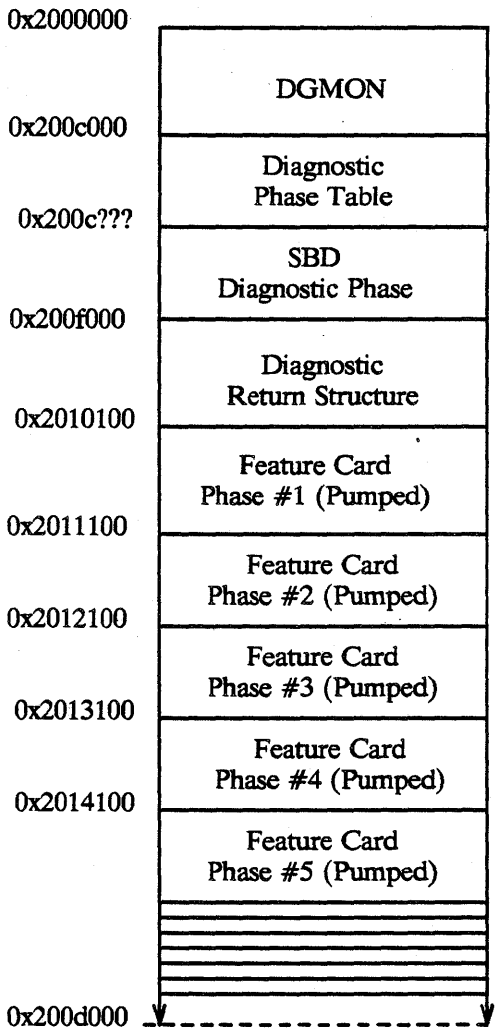


Figure B–5 System Board Diagnostic RAM Utilization for Pumped Cards

Refer to later sections of this appendix for more information on writing and compiling a C language source file to create diagnostic files. Before starting with code development, create a separate diagnostic floppy diskette for storing your work.

Putting Diagnostic Files on a Floppy Diskette

Diagnostic files should be created on a separate floppy diskette to minimize the possibility of deleting or corrupting valuable system files. Figure B-6 describes the commands required to make such a diskette that is bootable from firmware mode and mountable in multiuser mode.

```
# fmtflop -v /dev/rdsk/c0d0s6
# newboot /lib/olboot /lib/mboot
newboot: confirm request to write boot programs to /dev/rdsk/c0d0s7:y
# mkfs /dev/dsk/c0d0s5 1303 1 18
Mkfs: /dev/dsk/c0d0s5?
(DEL if wrong)
bytes per logical blocks = 1024
total logical blocks = 702
total inodes = 160
gap (physical blocks) = 1
cylinder size (physical blocks) = 18
mkfs: Available blocks = 689
# labelit /dev/rdsk/c0d0s5 dgn 060487
Current fsname: , Current volname: , Blocks: 1404, Inodes: 160
FS Units: 1KB, Date last mounted: date
NEW fsname = dgn, NEW volname = 060487 -- DEL if wrong!!
# mount /dev/dsk/c0d0s5 install
# find /demon /dgn /filledt -print | cpio -puvdm /install
/install/dgmon
/install/dgn/edt_data
/install/dgn/SBD
/install/dgn/X.SBD
/install/dgn/PORTS
/install/dgn/X.PORTS
/install/dgn/HR1
/install/dgn/X.HR1
442 blocks
# umount /dev/dsk/c0d0s5
```

Figure B–6 Making a Diagnostic Floppy Diskette

Organization of the Diagnostic Development Floppy

Figure B-7 shows the directories and files that should be included on the diagnostics development floppy. The floppy includes diagnostic files and the source for the diagnostics. The floppy can be mounted in the multiuser system and the programs (diagnostic phases) can be written, edited, and compiled using the standard UNIX system tools. Subsequently, the same floppy can be used as a source of diagnostic programs when a 3B2 computer is querying from the firmware mode. The "Compiling Diagnostic Phases" section in this appendix describes this in detail.

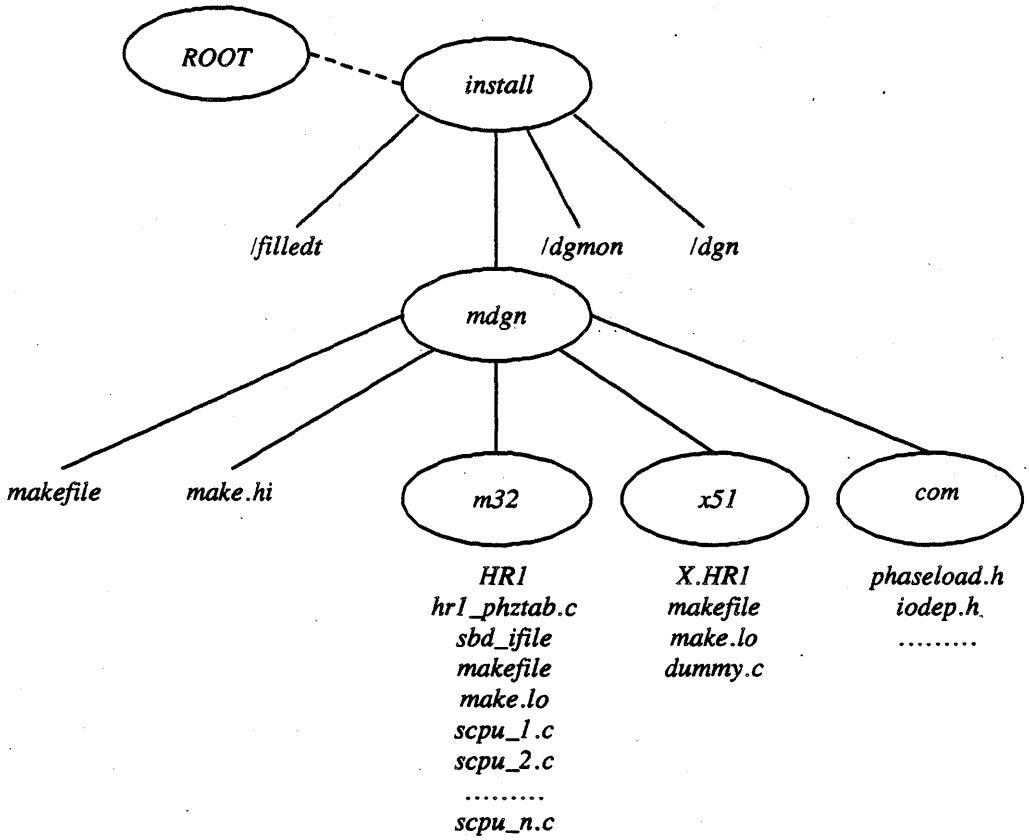


Figure B-7 Organization of the Diagnostics Development Floppy Disk

Diagnostics Source File Organization

Figure B-8 shows the organization of diagnostic files for HR1 feature card. The top directory, *mdgn*, contains three subdirectories

- *m32* — systems board diagnostics directory
- *x51* — feature card object code directory
- *com* — common header files directory

The *mdgn* directory also contains two makefiles, *makefile* and *make.hi*. From the *mdgn* directory, enter **make** to compile all of the subordinate diagnostic files.

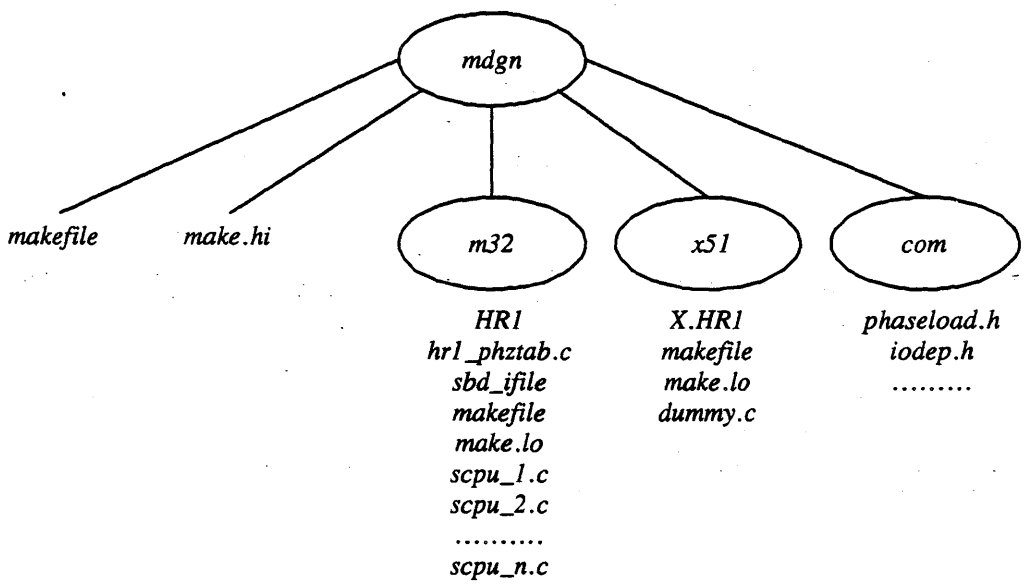


Figure B-8 mdgn Directory

A full listing of the HR1 diagnostic source is presented in the source code sections at the end of this appendix.

System Board Diagnostics Directory (m32)

The *m32* directory contains all the necessary files to generate the system board based diagnostics. The purpose and the functions of the individual programs in this directory can be summerized as follows

- The diagnostic monitor runs the diagnostic phases according to the phase table *hr1_phztab.c*.
- Individual diagnostic programs (phases) are in files *scpu_1.c, scpu_2.c, scpu_3.c,* and so on. These diagnostic programs interact with HR1 feature card, causing it to go through specified test phases.
- The individual diagnostic phases and the phase table is compiled according to rules stated in *makefile* and *make.lo* .
- The individual phases and the phase table is loaded into a 3B2 computer's main memory in accordance with *sbd_ifile*.
- Objects of the individual phases are combined into one HR1 file.

Feature Card Object Code Directory (x51)

The *x51* directory contains all the files necessary to generate feature card object code if this feature is selected. Because the diagnostic files for the HR1 feature card are stored in ROM, this directory contains only the files needed to compile the dummy file to satisfy **dgmon** requirements. This dummy file is assigned the name *X.HR1*. If the feature card can download programs into its memory (see Table B-3), objects of the individual phases are combined into a one file: *X.HR1*. In this case, the directory contains all the diagnostic phases to be downloaded into the feature card memory. These diagnostic phases are downloaded by the system board diagnostic phases. For example, systems board diagnostic phase *scpu_1*. downloads *scpu_1.c*, *scpu_2.c* downloads *scpu_2.c*, and *scpu_3.c* downloads *scpu_3.c*, and so on.

Common Header File Directory (com)

The *com* directory contains all the common header files. These header files contain definitions for generic feature cards as well as specific common I/O feature cards. Figure B-8 describes the files that should be in the *com* directory.

Diagnostic Phase Table

The diagnostic phase table is the first program loaded into main memory. All other diagnostic phases are loaded after the diagnostic phase table (a map that includes the load point for the diagnostic phase table is shown in Figure B-4). Figure B-9 lists a sample diagnostic phase table.

```
 1  /**
 2  * Copyright (c) 1986 AT&T
 3  *     - pb_phztab.c -
 4  *
 5  * Diagnostic phase table for -HR- Board
 6  **/

 7  #include <sys/firmware.h>
 8  #include <sys/diagnostic.h>

 9  extern unsigned char scpu_1(), scpu_2(), scpu_3(), scpu_4(), scpu_5();
10  extern unsigned char scpu_6(), scpu_7();
11
12  struct phtab phptr[] = {
13    {scpu_1, NORML, "Phase 1 - Init ID Int Register Check"},
14    {scpu_2, NORML, "Phase 2 - Parallel Port Out Test"},
15    {scpu_3, NORML, "Phase 3 - Serial Port Out  Check"},
16    {scpu_4, INTERACT, "Phase 4 - Serial Port In Check"},
17    {scpu_5, DEMAND, "Phase 5 - Memory Read / Write Test"},
18    {scpu_6, INTERACT, "Phase 6 - Parallel Port In Check"},
19    {scpu_7, DEMAND, "Phase 7 - dummy"},
20    {scpu_7, END, ""}
21  };
```

Figure B–9 Diagnostic Phase Table Example

As shown in Figure B-9 in lines 12 through 21, the diagnostic phase table structure contains three fields: the phase name, the phase type, and a description. For example, in line 13 the phase name is *scpu_1*, the phase type is NORML, and the description is "Phase 1 - Init ID Int Register Check."

If the phase type field is NORML (normal), the phase is executed by **dgmon** in noninteractive mode during autoboot. If the phase type field is DEMAND or INTERACT, the phase can only be run in the interactive mode of MCP. DEMAND indicates that the phase performs comprehensive diagnostics.

The interactive phase type (noted by the INTERACT phase type) requires operator interaction.

NOTE: The END phase type must be the last phase type specified. In addition, the END phase type should repeat the previously specified phase name and the description field must end with a period (".").

A Loader Option File

The loader option file is created to ensure that the diagnostic phase table is loaded into memory first, at address 0x200c00. In the example in Figure B-10, the loader option file is named *sbd_ifile* (this file is invoked by *makefile* in the *m32* directory).

```
1   /*
 2   * Copyright (c) 1986 AT&T                    - sbd_ifile -
 3   *
 4   * This file loads SBD diagnostic code.  The phase table must
 5   * be loaded first and must start at address 0x200c000.
 6   */
 7   MEMORY
 8   {
 9     PHZTBL:       origin = 0x200c000,     length = 0x70000
10   }
11   SECTIONS
12   {
13     .phztab:
14     {
15         _start = .;
16         hr1_phztab.o(.data)
17     } > PHZTBL
18     .text:
19     {
20     } > PHZTBL
21     .data:
22     {
23     } > PHZTBL
24     .bss:
25     {
26     } > PHZTBL
27   }
```

Figure B-10 Loader Option File Example

Diagnostic Phases

Figure B-11 is an example of a diagnostic phase for the HR1 feature card. This program tests to see if the HR1 feature card is able to read the identification code from the ID hardware register located on the feature card and tests the interrupt vector register.

```
 1   #include <sys/diagnostic.h>
 2   #include <sys/firmware.h>
 3   #include <sys/sbd.h>
 4   #include <sys/edt.h>
 5   #include <sys/cio_defs.h>
 6   #include <ciofw.h>
 7   #include <iodep.h>
 8   #include <sys/queue.h>
 9   #include <phaseload.h>
10   #include <per_dgn.h>
11   #include <ppc_dgn.h>

12   #define DEBUG

13   /**
14   *     - scpu_1() -
15   *
16   * Copyright (c) 1986 AT&T
17   *
18   * This routine starts the HR1 tests.
19   **/
```

Figure B-11 HR1 Diagnostic Phase (*part 1 of 4*)

```
20   struct dgnret dgnret;
21   char ph_no;
22   unsigned short etime;
23   int (*efunc)();
24   scpu_1()
25   {
26     register int i, j;
27     register int delay1 = 1000;
29     long dly1, save_int;
30     int pb_slot;      /* slot # of this board */
31     int vec_num;      /* interrupt vector number */
32     int ass_ID = 0x72;  /* assigned board's id */
33     int ID, VEC;      /* board's id */
34     char *pb_id;      /* id address */
35     char *pb_vec;     /* interrupt address */
36     char *pb_par;     /* parallel port address */
37     char *pb_sero;       /* serial out port address */
38     char *pb_seri;       /* serial in port address */

39     /* phase execution time */
40     unsigned short etime = 2;

41     /* global phase number */
42     ph_no = 1;

43     /* print test header */
44     PRINTF("HR1 Phase: %d   Name: SCPU_1   Type: NORMAL\n", ph_no);
45     PRINTF("Test Count: 1   Time: %d sec.\n", etime);

46     pb_slot = EDTP(OPTION)->opt_slot;  /* get board slot # from EDT */
```

Figure B–11 HR1 Diagnostic Phase (*part 2 of 4*)

```
47      /* calculate board access vectors */
48      pb_id = (char *)((pb_slot * 0x200000) + 0x1);    /* ID code reg*/
49      pb_seri = (char *)((pb_slot * 0x200000) + 0x5); /* serial in */
50      pb_vec = (char *)((pb_slot * 0x200000) + 0x7);  /* int vec loc */
51      pb_sero = (char *)((pb_slot * 0x200000) + 0xfe); /* serial out */
52      pb_par = (char *)((pb_slot * 0x200000) + 0xff);  * parallel port */

53    #ifdef   DEBUG
54      PRINTF("BOARD LOCATED IN SLOT  %d\n", pb_slot);
55    #endif

56    /* calculate vector number */
57      vec_num = pb_slot * 0x10;

58    /* Read the board's ID number back from the  ID register */
59      ID = *pb_id;
60      PRINTF("ID CODE = %x\n", ID);

61    /* Write vector number into vector register */
62      for (j = 0; j < delay1; j++);
63      *pb_vec = (char)vec_num;

64    /* Read the vector number back from the vector register */
65      for (j = 0; j < delay1; j++);
66      VEC = *pb_vec;
67      PRINTF("INTERRUPT VECTOR = %x\n", VEC);
```

Figure B–11 HR1 Diagnostic Phase (*part 3 of 4*)

```
68      if (ID != ass_ID)
69          {
70          PRINTF("\n\nID CODE = %x IT SHOULD BE %x \n", ID,ass_ID);
71          return(FAIL);
72          }
73      else if (VEC != vec_num)
74          {
75          PRINTF("\n\nVECTOR ID = %x IT SHOULD BE %x \n", VEC,vec_num);
76          return(FAIL);
77          }
78      else
79          return(PASS);

80    } /* end scpu_1 */
```

Figure B–11 HR1 Diagnostic Phase (*part 4 of 4*)

Diagnostic Template

A template should be used to maintain standardization between messages for normal and demand diagnostic phases. This template allows one 72-column line for each of the following

- phase title and type
- output of the warning messages and input directions
- time it should take for the phase to execute
- total number of times the phase executes

To comply with the above requirements, a test header should be printed using PRINTF(D8X) statements. The first PRINTF statement should identify the phase and its type. The second PRINTF statement should list the number of times and the time (in seconds) for the phase to execute.

Note that these messages are only displayed during interactive MCP mode when the phase number is specified. For example, if the following commands are entered in firmware mode

```
dgn hr1
```

Because I/O is turned off by the **dgmon**, no used messages are displayed. However, when the phase number is specified, PRINTF messages are displayed.

```
dgn hr1 ph=1
```

pb_slot

A call to the standard library functions (located in */usr/include/firmware.h*) is in the body of the source program for *scpu_1.c*. The code for this call is contained in line 44 of the program is provided at the end of this appendix.

```
44    pb_slot = EDTP(OPTION) ->opt_slot;
```

This statement generates a slot number for a 3B2 expansion bus in which the feature card to be diagnosed is located. The slot number permits calculation of the base address for the feature card.

This is possible because the feature card slots in a 3B2 expansion bus are assigned unique addresses as shown in Table B-5.

Table B−5 Physical Address Assignment on Expansion Slots

slot number	3B2 physical address
1	0x200000
2	0x400000
3	0x600000
4	0x800000
5	0xa00000
6	0xc00000
7	0xe00000
8	0x1000000
9	0x1200000
10	0x1400000
11	0x1600000
12	0x1800000
13	0x1A00000
14	0x1C00000
15	0x1E00000

From the base address of the feature card, all useful feature card addresses can be calculated. For the HR1 feature card, the following addresses are significant

Table B−6 HR1 Feature Card Usable Addresses

Address	Description
pb_id	HR1 feature card identification register
pb_vec	interrupt vector register
pb_par	parallel port (input and output)
pb_sero	serial port output
pb_seri	serial port input

In addition, there is also an address defined in the phase *scpu_5* for the beginning of the RAM on the HR1 feature card.

The phase *scpu_1* tests to see if the HR1 feature card can identify itself properly. The HR1 feature card phase provides an identification code when tested by the 3B2 computer. Also, the card has the ability to accept and present its interrupt vector.

PASS - FAIL

Control of pass-fail actions occurs by **return** statements sent back to **dgmon**. In the case of the HR1 feature card *scpu_1* phase, pass-fail is controlled by the statements in Figure B-12. Complete code for this phase is provided at the end of this appendix.

```
66    if (ID != ass_ID)
67       {
68       PRINTF("\n\nID CODE = %x IT SHOULD BE %x \n", ID,ass_ID);
69       return(FAIL);
70       }
71    else if (VEC != vec_num)
72       {
73       PRINTF("\n\nVECTOR ID = %x IT SHOULD BE %x \n", VEC,vec_num);
74       return(FAIL);
75       }
76    else
77       return(PASS);
```

Figure B–12 Pass-Fail Control Statements

In line 77, the **return(PASS)** statement causes **dgmon** to pass the phase. In lines 69 and 74, the **return(FAIL)** statements signals **dgmon** to fail the phase.

Compiling Diagnostic Phases

This section describes how to compile diagnostic phases on the 3B2 computer for feature cards. Included is the compile process for the HR1 feature card.

In the previous section on making a diagnostic floppy, a set of existing */dgn* diagnostic files were transferred onto a specially initialized floppy diskette. In addition to these copied files, you should create and populate the *mdgn* directory as shown in the Figure B-7 and Figure B-8. Finally, you need to populate the subdirectories with source code.

IMPORTANT: The compilation procedure that follows assumes that you have two 3B2 computers, one in firmware mode for the execution and testing of diagnostic code (computer #1), and one in multiuser mode to be used for compilation of diagnostics (computer #2).

Any new feature cards should be previously installed on the computer that is in firmware mode before starting the activities in this section.

The following procedure describes how to compile a diagnostic phase.

1 Put computer #1 into firmware mode by entering

shutdown -i5 -y -g0

2 At the FIRMWARE MODE message, enter the firmware password. If your computer displays a ">" prompt, enter

boot

3 Install the **mdgn** floppy in the floppy disk drive.

4 At the `Enter name of program to execute [ ]` prompt, enter

dgmon

5 Next, the system asks for the disk option, either hard disk (which is the default) or floppy disk (FD5), enter

FD5

6 The green light on the floppy disk drive illuminates and and about 45 seconds later the **dgmon** prompt appears.

7 Display the HR1 feature card diagnostic phases by entering

l hr1

8 Execute all the HR1 phases observing the HR1 feature card performance.

9 To do this step and the next step, change the phase source programs and recompile them.

Change the diagnostic phase #5 (memory read/write test for the HR1 feature card) to be a NORML phase. The phase should identify itself as such.

10 Write phase #7 for the HR1 feature card to be a demand type. This phase should write ten patterns of 0x0f and 0xf0 to the parallel output port. Each time a pattern is executed, a sequence number is displayed on the terminal (serial out) such as: 1, 2, 3, ... 10.

11 After computer #1 finishes executing **dgmon**, wait until the green light on the floppy disk drive illuminates and remove the floppy disk.

12 In computer #2, install the **mdgn** floppy and enter

mount /dev/dsk/c0d0s5

The green light on the floppy disk drive then illuminates.

CAUTION: Do not remove the floppy diskette from the drive until after executing step 16.

13 Change directory to */install/mdgn*.

14 Edit or create the appropriate code as needed.

15 Change directory to */install/mdgn* and enter **make**. The command recompiles all the affected files and remakes the diagnostic object file located in */install/dgn/HR1*.

In case of an error, edit the affected source files and repeat this step.

16 Change directory to root (/) and enter

umount /dev/dsk/c0d0s5

This unmounts the diskette. When the green light on the floppy disk drive goes out, remove the **mdgn** floppy from computer #2.

17 Insert the **mdgn** floppy in the computer #1 and execute the newly created phase.

Repeat steps 12 through 17 as needed.

The following sections list the source code for the programs previously explained in this appendix.

ppc_dgn.h

```
 1  /**
 2  *       - ppc_dgn.h -
 3  *
 4  *   Diagnostic information for the 3B2 ports board.
 5  **/

 6  /*
 7   *  memory boundaries
 8   */
 9  #define  T (unsigned int *)0x0000 /* Low RAM test range (16k) */
10  #define  LRAMEND (unsigned int *)0x3fff
11  #define  HRAMSTRT (unsigned int *)0x4000 /* High RAM test range (16k) */
12  #define  HRAMEND (unsigned int *)0x7fff
13  /*
14   *  peripheral rom test values
15   */
16  #define  ROMSTART (unsigned char *)0xfc000 /* 16k ROM */
17  #define  ROMCHKSM (unsigned char *)0xfffee /* checksum addr */
18  /*
19   *  SBD memory info
20   */
21  #define  PIOPAGE 2  /* page register value for PIO tests */
22  #define  SRMCSTRT (unsigned char *)0x80000 /* PIO byte start location */
23  #define  SRMISTRT (unsigned int *)0x80000 /* PIO int start location */
24  /*
25   *  DMA page register value
26   */
27  #define  DMAPAGE 0x03   /* use fifth page so we don't
28            overwrite the diagnostic code */
29  /*
30   *  Last SBD RAM address to use in PIO diagnostics
31   *      (pio_1.c, pio_2.c)
32   */
33  #define SRMCEND (unsigned char *)0x9ffff /* PIO byte end address */
34  #define SRMIEND (unsigned int *)0x9fffe   /* PIO int end address */
35  /*
36   *  interrupt vector returned to SBD
37   */
38  #define  INTVECT 0x3
39  /*
40   *  address offset to peripheral devices
```

```
41    */
42   #define  IO_BASE 0x600
43   /*
44    *  duart 0 addresses
45    */
46   #define D0_MR1_2A(IO_BASE + 0x00)
47   #define D0_A_SR_CSR  (IO_BASE + 0x02)
48   #define D0_A_CMND(IO_BASE + 0x04)
49   #define D0_A_DATA(IO_BASE + 0x06)
50   #define D0_IPC_ACR   (IO_BASE + 0x08)
51   #define D0_IS_IMR(IO_BASE + 0x0a)
52   #define D0_CTUR   (IO_BASE + 0x0c)
53   #define D0_CTLR   (IO_BASE + 0x0e)
54   #define D0_MR1_2B(IO_BASE + 0x10)
55   #define D0_B_SR_CSR  (IO_BASE + 0x12)
56   #define D0_B_CMND(IO_BASE + 0x14)
57   #define D0_B_DATA(IO_BASE + 0x16)
58   #define D0_IP_OPCR   (IO_BASE + 0x1a)
59   #define  D0_SCC_SOPBC    (IO_BASE + 0x1c)
60   #define  D0_SCC_ROPBC    (IO_BASE + 0x1e)
61   /*
62    *  duart 1 addresses
63    */
64   #define D1_MR1_2A(IO_BASE + 0x80)
65   #define D1_A_SR_CSR  (IO_BASE + 0x82)
66   #define D1_A_CMND(IO_BASE + 0x84)
67   #define D1_A_DATA(IO_BASE + 0x86)
68   #define D1_IPC_ACR   (IO_BASE + 0x88)
69   #define D1_IS_IMR(IO_BASE + 0x8a)
70   #define D1_CTUR   (IO_BASE + 0x8c)
71   #define D1_CTLR   (IO_BASE + 0x8e)
72   #define D1_MR1_2B(IO_BASE + 0x90)
73   #define D1_B_SR_CSR  (IO_BASE + 0x92)
74   #define D1_B_CMND(IO_BASE + 0x94)
75   #define D1_B_DATA(IO_BASE + 0x96)
76   #define D1_IP_OPCR   (IO_BASE + 0x9a)
77   #define  D1_SCC_SOPBC    (IO_BASE + 0x9c)
78   #define  D1_SCC_ROPBC    (IO_BASE + 0x9e)
79   /*
80    *  duart control variables
81    */
82   #define RSTMRPT   0x10
83   #define INT7BT    0x12
84   #define INT8BT    0x13
85   #define EXT7BT    0x12
86   #define EXT8BT    0x13
87   #define INTLP 0x8f
```

```
 88  #define EXTLP 0x0f
 89  #define BAUDA 0x44   /* 300 baud */
 90  #define BAUDB 0x66   /* 1200 baud */
 91  #define BAUDC 0x99   /* 4800 baud */
 92  #define BAUDD 0xbb   /* 9600 baud */
 93  #define BAUDE 0xcc   /* 19.2K baud */
 94  /*
 95   *  duart status variables
 96   */
 97  #define TXRDY0    (*STATRG0 & 0x04)
 98  #define RXRDY0    (*STATRG0 & 0x01)
 99  #define TXRDY1    (*STATRG1 & 0x04)
100  #define RXRDY1    (*STATRG1 & 0x01)
101  #define TXRDY2    (*STATRG2 & 0x04)
102  #define RXRDY2    (*STATRG2 & 0x01)
103  #define TXRDY3    (*STATRG3 & 0x04)
104  #define RXRDY3    (*STATRG3 & 0x01)
105  #define FFULL0    (*STATRG0 & 0x02)
106  #define FFULL1    (*STATRG1 & 0x02)
107  #define FFULL2    (*STATRG2 & 0x02)
108  #define FFULL3    (*STATRG3 & 0x02)
109  #define OVRRUN0 (*STATRG0 & 0x10)
110  #define OVRRUN1 (*STATRG1 & 0x10)
111  #define OVRRUN2 (*STATRG2 & 0x10)
112  #define OVRRUN3 (*STATRG3 & 0x10)
113  /*
114   *  printer addresses
115   */
116  #define PORTA (IO_BASE + 0x100)
117  #define PORTC (IO_BASE + 0x101)
118  /*
119   *  printer status variables
120   */
121  #define PRBUSY    (*PORTC & 0x10)
122  #define PRPE   (*PORTC & 0x20)
123  #define PRSEL (*PORTC & 0x40)
124  #define PRFALT    (*PORTC & 0x80)
125  #define PRREST    (*PORTC & 0x01)
126  #define PRSTRB    (*PORTC & 0x02)
127  #define PRAUTF    (*PORTC & 0x04)
128  /*
129   *  test variables
130   */
131  #define  SHORTZERO   0x0000
132  #define  BYTEZERO 0x00
133  #define  SHORTONES   0xffff
134  #define  BYTEONES 0xff
```

```
135   #define  SHORTAOAZ   0xaaaa
136   #define  BYTEAOAZ 0xaa
137   #define  SHORTAZAO   0x5555
138   #define  BYTEAZAO 0x55
```

ciofw.h

```
 1   /*
 2    *              Copyright 1984 AT&T
 3    *      This header file contains declarations and defines
 4    *      those which are used by the common I/O routines only.
 5    */
 6
 7   #define MAX_XFER  0x400     /* max bytes XFERd by movoffb&movoffbw */
 8   #define CLR_BRQ   0x2000004 /* addr to write BDID - clear bus reqs */
 9   #define DPD_OFFS  0x80000   /* DPD RAM offset                      */
10   #define UMCS      0xfc38    /* value for upper memory chip select  */
11   #define LMCS      0x3ff8    /* value for lower memory chip select  */
12   #define MMCS      0x8000    /* value for middle memory chip select */
13   #ifdef MEMSPACE
14   #       define PACS  0xc03a     /* value for PACS register         */
15   #       define MPCS  0xa0f8     /* value for memory block size     */
16   #else
17   #       define PACS  0x7a       /* value for PACS register         */
18   #       define MPCS  0xa0b8     /* value for memory block size     */
19   #endif
20   #define FULL   0x0              /* value for queue full in putcomp */
21   #define DMA_CWB   0xb6ae        /* DMA cntrl word val to xfer bytes*/
22   #define DMA_CWW   0xb6af        /* DMA cntrl word val to xfer words*/
23   #define INT0MSK   0x10          /* mask value for INT 0            */
24   #define INT1MSK   0x20          /* mask value for INT 1            */
25   #define RQ        1             /* request queue                   */
26   #define CQ        0             /* completion queue                */
27   /*
28     This file is included by both 'C' language source and assembly
29     language source.  The assembly code does not wish to see the
30     'C' specific stuff, and so it defines a macro named "ASSY".
31   */
32
33   #ifndef ASSY
34           typedef struct cmds{
35                   char    opcode;
36                   short (*func)();
37           }CMDS;
38   #endif
```

cio_dev.h

```
/*
                    Copyright 1984 AT&T

          This file contains macros for accessing the various IAPX186
          devices, located in I/O space or memory space, depending upon
          how one compiles the common I/O.

          The following are the base locations of the various locations
          within the I/O (or memory) spectrum.
*/
#ifdef MEMSPACE

#define CHAR(x)          *((char *)x)
#define SHORT(x)         *((short *)x)
#define USHORT(x)        *((unsigned short *)x)
#define LONG(x)          *((long *)x)
#define ULONG(x)         *((unsigned long *)x)

#define I        0xc0400          /* internal register space */
#define X        0xc0000          /* external register space */
#else
#define I        0xff00           /* internal register space */
#define X        0x0400           /* external register space */

#endif
/*
      The following section comes in two versions: one for C
      programs and one for assembly language programs.  The only
      difference is the convention for expression inclusion: C
      uses parentheses and the assembler uses square brackets.
      If you change data in one area, BE SURE TO CHANGE THE
      CORRESPONDING DATA IN THE OTHER.
*/

#ifdef ASSY

#define IC    [I+0x20]    /* Interrupt controller control regs */
#define T0    [I+0x50]    /* Timer 0 control registers */
#define T1    [I+0x58]    /* Timer 1 control registers */
#define T2    [I+0x60]    /* Timer 2 control registers */
#define CS    [I+0xa0]    /* Chip Select control registers */
#define D0    [I+0xc0]    /* DMA 0 control registers */
```

```
36  #define D1      [I+0xd0]      /* DMA 1 control registers */

37  /*
38          Interrupt Controller control registers
39  */

40  #define IC_EOI          [IC+0x2]     /* end of interrupt */
41  #define IC_POLL         [IC+0x4]     /* poll */
42  #define IC_PSTAT        [IC+0x6]     /* poll status */
43  #define IC_MASK         [IC+0x8]     /* mask */
44  #define IC_PMASK        [IC+0xa]     /* priority mask */
45  #define IC_INSVC        [IC+0xc]     /* in-service */
46  #define IC_IREQ         [IC+0xe]     /* interrupt request */
47  #define IC_ISTAT        [IC+0x10]    /* interrupt status */
48  #define IC_TCTRL        [IC+0x12]    /* timer control */
49  #define IC_DMA0         [IC+0x14]    /* DMA 0 */
50  #define IC_DMA1         [IC+0x16]    /* DMA 1 */
51  #define IC_INT0         [IC+0x18]    /* interrupt 0 */
52  #define IC_INT1         [IC+0x1a]    /* interrupt 1 */
53  #define IC_INT2         [IC+0x1c]    /* interrupt 2 */
54  #define IC_INT3         [IC+0x1e]    /* interrupt 3 */

55  /*
56          The following are areas of I/O space used to
57          control the timers.
58  */

59  #define T0_COUNT        [T0+0x0]     /* count */
60  #define T0_MCA          [T0+0x2]     /* max count a */
61  #define T0_MCB          [T0+0x4]     /* mas count b */
62  #define T0_MODE         [T0+0x6]     /* count register */

63  #define T1_COUNT        [T1+0x0]     /* count */
64  #define T1_MCA          [T1+0x2]     /* max count a */
65  #define T1_MCB          [T1+0x4]     /* mas count b */
66  #define T1_MODE         [T1+0x6]     /* count register */

67  #define T2_COUNT        [T2+0x0]     /* count */
68  #define T2_MCA          [T2+0x2]     /* max count a */
69  #define T2_MODE         [T2+0x6]     /* count register */

70  /*
71          The following define the control area for the
72          chip select registers.
73  */

74  #define CS_UM           [CS+0x0]     /* upper memory          */
```

```
 75   #define CS_LM          [CS+0x2]    /* lower memory          */
 76   #define CS_PA          [CS+0x4]    /* PACS register         */
 77   #define CS_MM          [CS+0x6]    /* middle memory         */
 78   #define CS_MP          [CS+0x8]    /* memory block size     */

 79   /*
 80           The following the control space of the DMA units
 81   */

 82   #define D0_SRCL        [D0+0x0]    /* source lower 16 bits */
 83   #define D0_SRCH        [D0+0x2]    /* source upper 4 bits */
 84   #define D0_DESTL       [D0+0x4]    /* destination lower 16 bits */
 85   #define D0_DESTH       [D0+0x6]    /* destination upper 4 bits */
 86   #define D0_TCOUNT      [D0+0x8]    /* transfer count*/
 87   #define D0_CTRL        [D0+0xa]    /* DMA unit zero control word */

 88   #define D1_SRCL        [D1+0x0]    /* source lower 16 bits */
 89   #define D1_SRCH        [D1+0x2]    /* source upper 4 bits */
 90   #define D1_DESTL       [D1+0x4]    /* destination lower 16 bits */
 91   #define D1_DESTH       [D1+0x6]    /* destination upper 4 bits */
 92   #define D1_TCOUNT      [D1+0x8]    /* transfer count*/
 93   #define D1_CTRL        [D1+0xa]    /* DMA unit one control word */

 94   /*
 95           The following define the space of the off-chip
 96           registers located on the peripheral board.
 97   */

 98   #define CLRINT0        [X+0x88]    /* reset int0 latch */
 99   #define CLRINT1        [X+0x89]    /* reset int1 latch */
100   #define CLRINT2        [X+0x8a]    /* reset int2 latch */
101   #define CLRINT3        [X+0x8b]    /* reset int3 latch */
102   #define ID_16          [X+0x80]    /* 16-bit ID register */
103   #define INTV_ID        [X+0x81]    /* interrupt vector ID reg */
104   #define PAGE_REG       [X+0x82]    /* page register */
105   #define PCSR_REG       [X+0x84]    /* PCSR register    */
106   #define BAF_BIT        [X+0x8e]    /* bus abort feature */
107   #define SYS_INT        [X+0x8f]    /* system interrupt */

108   #else

109   #define IC     (I+0x20)     /* Interrupt Controller control regs */
110   #define T0     (I+0x50)     /* Timer 0 control registers */
111   #define T1     (I+0x58)     /* Timer 1 control registers */
112   #define T2     (I+0x60)     /* Timer 2 control registers */
113   #define CS     (I+0xa0)     /* Chip Select control registers */
114   #define D0     (I+0xc0)     /* DMA 0 control registers */
```

```
115   #define D1    (I+0xd0)    /* DMA 1 control registers */
116   /*
117           Interrupt Controller control registers
118   */

119   #define IC_EOI          (IC+0x2)    /* end of interrupt */
120   #define IC_POLL         (IC+0x4)    /* poll */
121   #define IC_PSTAT        (IC+0x6)    /* poll status */
122   #define IC_MASK         (IC+0x8)    /* mask */
123   #define IC_PMASK        (IC+0xa)    /* priority mask */
124   #define IC_INSVC        (IC+0xc)    /* in-service */
125   #define IC_IREQ         (IC+0xe)    /* interrupt request */
126   #define IC_ISTAT        (IC+0x10)   /* interrupt status */
127   #define IC_TCTRL        (IC+0x12)   /* timer control */
128   #define IC_DMA0         (IC+0x14)   /* DMA 0 */
129   #define IC_DMA1         (IC+0x16)   /* DMA 1 */
130   #define IC_INT0         (IC+0x18)   /* interrupt 0 */
131   #define IC_INT1         (IC+0x1a)   /* interrupt 1 */
132   #define IC_INT2         (IC+0x1c)   /* interrupt 2 */
133   #define IC_INT3         (IC+0x1e)   /* interrupt 3 */

134   /*
135           The following are areas of I/O space used to
136           control the timers.
137   */

138   #define T0_COUNT        (T0+0x0)    /* count */
139   #define T0_MCA          (T0+0x2)    /* max count a */
140   #define T0_MCB          (T0+0x4)    /* mas count b */
141   #define T0_MODE         (T0+0x6)    /* count register */

142   #define T1_COUNT        (T1+0x0)    /* count */
143   #define T1_MCA          (T1+0x2)    /* max count a */
144   #define T1_MCB          (T1+0x4)    /* mas count b */
145   #define T1_MODE         (T1+0x6)    /* count register */

146   #define T2_COUNT        (T2+0x0)    /* count */
147   #define T2_MCA          (T2+0x2)    /* max count a */
148   #define T2_MODE         (T2+0x6)    /* count register */

149   /*
150           The following define the control area for the
151           chip select registers.
152   */

153   #define CS_UM           (CS+0x0)    /* upper memory          */
154   #define CS_LM           (CS+0x2)    /* lower memory          */
```

```
155   #define CS_PA          (CS+0x4)   /* PACS register          */
156   #define CS_MM          (CS+0x6)   /* middle memory          */
157   #define CS_MP          (CS+0x8)   /* memory block size      */

158   /*
159           The following the control space of the DMA units
160   */

161   #define D0_SRCL        (D0+0x0)   /* source lower 16 bits */
162   #define D0_SRCH        (D0+0x2)   /* source upper 4 bits */
163   #define D0_DESTL       (D0+0x4)   /* destination lower 16 bits */
164   #define D0_DESTH       (D0+0x6)   /* destination upper 4 bits */
165   #define D0_TCOUNT      (D0+0x8)   /* transfer count*/
166   #define D0_CTRL        (D0+0xa)   /* DMA unit zero control word */

167   #define D1_SRCL        (D1+0x0)   /* source lower 16 bits */
168   #define D1_SRCH        (D1+0x2)   /* source upper 4 bits */
169   #define D1_DESTL       (D1+0x4)   /* destination lower 16 bits */
170   #define D1_DESTH       (D1+0x6)   /* destination upper 4 bits */
171   #define D1_TCOUNT      (D1+0x8)   /* transfer count*/
172   #define D1_CTRL        (D1+0xa)   /* DMA unit one control word */

173   /*
174           The following define the space of the off-chip
175           registers located on the peripheral board.
176   */

177   #define CLRINT0        (X+0x88)   /* reset int0 latch */
178   #define CLRINT1        (X+0x89)   /* reset int1 latch */
179   #define CLRINT2        (X+0x8a)   /* reset int2 latch */
180   #define CLRINT3        (X+0x8b)   /* reset int3 latch */
181   #define ID_16          (X+0x80)   /* 16-bit ID register */
182   #define INTV_ID        (X+0x81)   /* interrupt vector id reg */
183   #define PAGE_REG       (X+0x82)   /* page register */
184   #define PCSR_REG       (X+0x84)   /* PCSR register    */
185   #define BAF_BIT        (X+0x8e)   /* bus abort feature */
186   #define SYS_INT        (X+0x8f)   /* system interrupt */
187   #endif
```

m a k e . l o

```
1     ########
2     #
3     # Copyright (c) 1986 AT&T
4     #
5     # make.lo for x51 side of HR1 diagnostics
6     #
7     ########
8     TITLE = makefile (x51 make.lo) for x51 side of hR1 Diagnostics
9     MACHINE = m32
10    DEFS = -Dm32
11    CFLAGS =
12
13    all:
14
15    d=`pwd`; echo "\n Now in $$d directory \n";
16
17    SRC =dummy.c
18
19    OBJ =dummy.o
20
21    PRODUCTS = X.HR1
22
23    $(PRODUCTS): $(OBJ)
24      $(LD) -o $(PRODUCTS) $(OBJ)
25      cp $(PRODUCTS) $(ROOT)/install/dgn/$(PRODUCTS)
26      $(STRIP) $(PRODUCTS)
27
28    .PRECIOUS: $(PRODUCTS)
29
30    #install:   all
```

m a k e f i l e

```
1   all: X.HR1
2     cc dummy.c  X.HR1
```

sbd_ifile

```
1   /*
2    *          - sbd_ifile -
3    *
4    *  This file is used to load the SBD diagnostic initialization
5    *  code.  The order is critical in that the phase table must
6    *  be the first thing loaded and must start at 0x200c000.
7    */
8   MEMORY
9   {
10   PHZTBL:       origin = 0x200c000,     length = 0x70000
11  }

12  SECTIONS
13  {
14   .phztab:
15   {
16       _start = .;
17       hr1_phztab.o(.data)
18   } > PHZTBL

19   .text:
20   {
21   } > PHZTBL

22   .data:
23   {
24   } > PHZTBL

25   .bss:
26   {
27   } > PHZTBL
28  }
```

hr1_phztab.c

```
 1   /**
 2   *
 3   *    Copyright (c) 1986 AT&T
 4   *
 5   *    Diagnostic phase table for -HR- Board
 6   **/

 7   #include <sys/firmware.h>
 8   #include <sys/diagnostic.h>

 9   extern unsigned char scpu_1(), scpu_2(), scpu_3(), scpu_4();
10   extern unsigned char scpu_5(), scpu_6(), scpu_7();

11   struct phtab phptr[] = {
12    {scpu_1, NORML, "Phase 1 - Init ID Int Register Check "},
13    {scpu_2, NORML, "Phase 2 - Parallel Port Out Test"},
14    {scpu_3, NORML, "Phase 3 - Serial Port Out  Check"},
15    {scpu_4, INTERACT, "Phase 4 - Serial Port In Check"},
16    {scpu_5, DEMAND, "Phase 5 - Memory Read / Write Test"},
17    {scpu_6, INTERACT, "Phase 6 - Parallel Port In Check"},
18    {scpu_7, DEMAND, "Phase 7 - dummy"},
19    {scpu_7, END, ""}
20   };
```

scpu_1.c

```
#include   <sys/diagnostic.h>
#include   <sys/firmware.h>
#include   <sys/sbd.h>
#include   <sys/edt.h>
#include   <sys/cio_defs.h>
#include   <ciofw.h>
#include   <iodep.h>
#include   <sys/queue.h>
#include   <phaseload.h>
#include   <per_dgn.h>
#include   <ppc_dgn.h>

#define DEBUG

/**
*
*   Copyright (c) 1986 AT&T
*
*   This routine starts the HR1 tests.
**/

struct dgnret dgnret;
char ph_no;
unsigned short etime;
scpu_1()
{
 register int i, j;
 register int delay1 = 1000;
 long dly1, save_int;
 int pb_slot;     /* slot # of this board */
 int vec_num;     /* interrupt vector number */
 int ass_ID = 0x72;  /* assigned board's id */
 int ID, VEC;     /* board's id */
 char *pb_id;     /* id address */
 char *pb_vec;    /* interrupt address */
 char *pb_par;    /* parallel port address */
 char *pb_sero;       /* serial out port address */
 char *pb_seri;       /* serial in port address */

 /* phase execution time */

 unsigned short etime = 2;
```

```
38    /* global phase number */

39    ph_no = 1;
40
41    /* print test header */

42    PRINTF("HR1 Phase: %d   Name: SCPU_1   Type: NORMAL\n", ph_no);
43    PRINTF("Test Count: 1   Time: %d sec.\n", etime);


44    pb_slot = EDTP(OPTION)->opt_slot;  /* get board slot # from edt */

45    /* calculate board access vectors */

46    pb_id = (char *)((pb_slot * 0x200000) + 0x1);    /* id code regist.*/
47    pb_seri = (char *)((pb_slot * 0x200000) + 0x5);    /* serial in */
48    pb_vec = (char *)((pb_slot * 0x200000) + 0x7);     /* int vec loc */
49    pb_sero = (char *)((pb_slot * 0x200000) + 0xfe);    /* serial out */
50    pb_par = (char *)((pb_slot * 0x200000) + 0xff);    /* parallel port */

51   #ifdef DEBUG
52    PRINTF("BOARD LOCATED IN SLOT  %d\n", pb_slot);
53   #endif

54   /* calculate vector number */
55    vec_num = pb_slot * 0x10;

56   /* Read the board's ID number back from the  ID register */
57    ID = *pb_id;
58    PRINTF("ID CODE = %x\n", ID);

59   /* Write vector number into vector register */
60    for (j = 0; j < delay1; j++);
61    *pb_vec = (char)vec_num;

62   /* Read the vector number back from the vector register */
63    for (j = 0; j < delay1; j++);
64    VEC = *pb_vec;
65    PRINTF("INTERRUPT VECTOR = %x\n", VEC);


66    if (ID != ass_ID)
67       {
68       PRINTF("\n\nID CODE = %x IT SHOULD BE %x \n", ID,ass_ID);
69       return(FAIL);
70       }
```

```
71    else if (VEC != vec_num)
72        {
73        PRINTF("\n\nVECTOR ID = %x IT SHOULD BE %x \n", VEC,vec_num);
74        return(FAIL);
75        }
76    else
77        return(PASS);


78    } /* end scpu_1 */
```

scpu_2.c

```
 1  #include   <sys/diagnostic.h>
 2  #include   <sys/firmware.h>
 3  #include   <sys/sbd.h>
 4  #include   <sys/edt.h>
 5  #include   <sys/cio_defs.h>
 6  #include   <ciofw.h>
 7  #include   <iodep.h>
 8  #include   <sys/queue.h>
 9  #include   <phaseload.h>
10  #include   <per_dgn.h>
11  #include   <ppc_dgn.h>

12  #define DEBUG

13  /* Byte pattern to be used to test parallel out port */

14  static char a[]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,
15       0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01,
16       0xff,0x11,0xff,0x22,0xff,0x44,0xff,0x88,0};

17  /**
18  *   Copyright (c) 1986 AT&T
19  *
20  *   This routine tests the "parallel out" port of the HR1 tests.
21  **/

22  struct dgnret dgnret;
23  extern char ph_no;
24  unsigned short etime;
25  scpu_2()
26  {
27   register int i, j;
28   register int delay1 = 20000;
29   long dly1, save_int;
30   int pb_slot;     /* slot # of this board */
31   int vec_num;     /* interrupt vector number */
32   char *pb_id;     /* ID address */
33   char *pb_vec;    /* interrupt address */
34   char *pb_par;    /* parallel port address */
35   char *pb_sero;      /* serial out port address */
36   char *pb_seri;      /* serial in port address */
37   char *p;
```

```
38    unsigned short etime = 2;   /* phase execution time */
39    ph_no = 2;        /* global phase number */

40    /* print test header */

41    PRINTF("HR1 Phase: %d   Name: SCPU_2   Type: NORMAL\n", ph_no);
42    PRINTF("Test Count: 3   Time: %d sec.\n", etime);

43    /* execute onboard diagnostic */

44    pb_slot = EDTP(OPTION)->opt_slot;  /* get board slot # from edt */

45    /* calculate board access vectors */

46    pb_id = (char *)((pb_slot * 0x200000) + 0x1);     /* ID code regist.*/
47    pb_seri = (char *)((pb_slot * 0x200000) + 0x5);     /* serial in */
48    pb_vec = (char *)((pb_slot * 0x200000) + 0x7);      /* int vec loc */
49    pb_sero = (char *)((pb_slot * 0x200000) + 0xfe);     /* serial out */
50    pb_par = (char *)((pb_slot * 0x200000) + 0xff);     /* parallel port */

51    PRINTF("PARALLEL PORT TEST\n");

52    /* Parallel out test */

53        for(i=0; i < 5; i++)
54    {
55       p = a;
56       while (*p != 0)
57       {
58          for(j=0; j < delay1; j++);
59          *pb_par = *p++;

60       } /* end while */
61    } /* end for */


62    return(PASS);


63   } /* end scpu_2 */
```

scpu_3.c

```
 1   #include   <sys/diagnostic.h>
 2   #include   <sys/firmware.h>
 3   #include   <sys/sbd.h>
 4   #include   <sys/edt.h>
 5   #include   <sys/cio_defs.h>
 6   #include   <ciofw.h>
 7   #include   <iodep.h>
 8   #include   <sys/queue.h>
 9   #include   <phaseload.h>
10   #include   <per_dgn.h>
11   #include   <ppc_dgn.h>

12   #define DEBUG



13   /**
14   *
15   *   Copyright (c) 1986 AT&T
16   *
17   *   This routine tests "serial out" port of HR1
18   **/

19   struct dgnret dgnret;
20   extern char ph_no;
21   unsigned short etime;
22   scpu_3()
23   {
24    register char *p;
25    register int j;
26    register int delay1 = 10000;
27    long dly1, save_int;
28    int pb_slot;      /* slot # of this board */
29    int vec_num;      /* interrupt vector number */
30    char *pb_id;      /* ID address */
31    char *pb_vec;     /* interrupt address */
32    char *pb_par;     /* parallel port address */
33    char *pb_sero;       /* serial out port address */
34    char *pb_seri;       /* serial in port address */

35    /* phase execution time */
```

```
36    unsigned short etime = 2;

37    /* global phase number */

38    ph_no = 3;

39    /* print test header */

40    PRINTF("HR1 Phase: %d   Name: SCPU_3   Type: NORMAL\n", ph_no);
41    PRINTF("Test Count: 3   Time: %d sec.\n", etime);

42    /* execute onboard diagnostic */

43    pb_slot = EDTP(OPTION)->opt_slot;  /* get board slot # from edt */

44    /* calculate board access vectors */

45    pb_id = (char *)((pb_slot * 0x200000) + 0x1);     /* ID code regist.*/
46    pb_seri = (char *)((pb_slot * 0x200000) + 0x5);     /* serial in */
47    pb_vec = (char *)((pb_slot * 0x200000) + 0x7);      /* int vec loc */
48    pb_sero = (char *)((pb_slot * 0x200000) + 0xfe);     /* serial out */
49    pb_par = (char *)((pb_slot * 0x200000) + 0xff);     /* parallel port */

50    PRINTF("\nSERIAL OUT PORT TEST\n");


51    /* Serial out test */


52    p="\n\r******* Serial Port Output Test ********\n\r";
53        while (*p != '\0')
54    {
55       for(j=0; j < delay1; j++);
56       *pb_sero = *p++;
57    }



58    return(PASS);


59   } /* end scpu_3 */
```

scpu_4.c

```
#include   <sys/diagnostic.h>
#include   <sys/firmware.h>
#include   <sys/sbd.h>
#include   <sys/edt.h>
#include   <sys/cio_defs.h>
#include   <ciofw.h>
#include   <iodep.h>
#include   <sys/queue.h>
#include   <phaseload.h>
#include   <per_dgn.h>
#include   <ppc_dgn.h>

#define DEBUG



/**
*
*   Copyright (c) 1986 AT&T
*
*   This routine tests serial in port of HR1
**/

struct dgnret dgnret;
extern char ph_no;
unsigned short etime;
scpu_4()
{
 register int i, j;
 register int delay1 = 30000;
 long dly1, save_int;
 int pb_slot;     /* slot # of this board */
 int vec_num;     /* interrupt vector number */
 char *pb_id;     /* ID address */
 char *pb_vec;    /* interrupt address */
 char *pb_par;    /* parallel port address */
 char *pb_sero;      /* serial out port address */
 char *pb_seri;      /* serial in port address */
 char byte1;
 char byte2;

 /* phase execution time */
```

```
37    unsigned short etime = 2;

38    /* global phase number */

39    ph_no = 4;
40
41    /* print test header */

42    PRINTF("HR1 Phase: %d   Name: SCPU_4   Type: NORMAL\n", ph_no);
43    PRINTF("Test Count: 3   Time: %d sec.\n", etime);

44    /* execute onboard diagnostic */

45    pb_slot = EDTP(OPTION)->opt_slot;  /* get board slot # from edt */

46                  /* calculate board access vectors */

47    pb_id = (char *)((pb_slot * 0x200000) + 0x1);     /* ID code regist.*/
48    pb_seri = (char *)((pb_slot * 0x200000) + 0x5);     /* serial in */
49    pb_vec = (char *)((pb_slot * 0x200000) + 0x7);      /* int vec loc */
50    pb_sero = (char *)((pb_slot * 0x200000) + 0xfe);     /* serial out */
51    pb_par = (char *)((pb_slot * 0x200000) + 0xff);     /* parallel port */

52    PRINTF("SERIAL IN PORT TEST\r\n");
53    PRINTF("BEGIN TYPING WHEN YOU HEAR BELLS AND 'GO' IS DISPLAYED\n\n");
54    for(j=0; i < delay1; j++);
55    for(j=0; i < delay1; j++);
56    PRINTF("GGGGGGO!!!!!\n\n");

57   /* Serial in  test */


58    for(i=0; i < 100; i++)
59        {
60        PRINTF("%c",*pb_seri);
61        for(j=0; j < delay1; j++);
62        }

63    return(PASS);


64    } /* end scpu_4 */
```

scpu_5.c

```
#include   <sys/diagnostic.h>
#include   <sys/firmware.h>
#include   <sys/sbd.h>
#include   <sys/edt.h>
#include   <sys/cio_defs.h>
#include   <ciofw.h>
#include   <iodep.h>
#include   <sys/queue.h>
#include   <phaseload.h>
#include   <per_dgn.h>
#include   <ppc_dgn.h>

#define DEBUG



/**
*   Copyright (c) 1986 AT&T
*
*      This routine tests READ/WRITE capabilities
*   of onboard RAM of HR1
**/

struct dgnret dgnret;
extern char ph_no;
unsigned short etime;
scpu_5()
{
 register int i, j;
 register int delay1 = 1000;
 int pb_slot;     /* slot # of this board */
 int vec_num;     /* interrupt vector number */
 int ram_size = 0x61;    /* 8751 ram size -9 */
 char *pb_id;     /* ID address */
 char *pb_vec;    /* interrupt address */
 char *pb_sram;       /* start of ram */
 char *pb_par;    /* parallel port address */
 char *pb_sero;       /* serial out port address */
 char *pb_seri;       /* serial in port address */
 char wbyte1 = 0x55, wbyte2 = 0xaa;  /* bytes with */
                 /* which RAM is tested */
 char rbyte;              /* byte with which RAM is tested */
```

```
38    /* phase execution time */

39    unsigned short etime = 10;

40    /* global phase number */

41    ph_no = 5;

42    /* print test header */

43    PRINTF("\n\r\nHR1 Phase: %d   Name: SCPU_5   Type: DEMAND\n", ph_no);
44    PRINTF("Test Count: 6   Time: %d sec.\n", etime);

45    /* execute onboard diagnostics */

46    pb_slot = EDTP(OPTION)->opt_slot;  /* get board slot # from edt */

47    /* calculate board access vectors */

48    pb_id = (char *)((pb_slot * 0x200000) + 0x1);     /* ID code regist.*/
49    pb_seri = (char *)((pb_slot * 0x200000) + 0x5);     /* serial in */
50    pb_vec = (char *)((pb_slot * 0x200000) + 0x7);      /* int vec loc */
51    pb_sram = (char *)((pb_slot * 0x200000) + 0x9);      /* start ram loc */
52    pb_sero = (char *)((pb_slot * 0x200000) + 0xfe);     /* serial out */
53    pb_par = (char *)((pb_slot * 0x200000) + 0xff);     /* parallel port */

54    PRINTF("\n\rON BOARD READ/WRITE RAM TEST \r\n");


55    for(i = 0; i < ram_size; i++)     /* ram_size -9 */
56    {
57       *(pb_sram + i) = wbyte1;   /* write first pattern */
58       for(j=0; j < delay1; j++);
59       rbyte = *(pb_sram + i);    /* read first time */
60       for(j=0; j < delay1; j++);
61       rbyte = *(pb_sram + i);    /* read second time */

62       PRINTF("%x", rbyte);       /* display the read back byte */
63       if (rbyte != wbyte1)
64       {
65          PRINTF("\n\r LOCATION %xh FAILED! READ %xh SHOULD READ %xh\n\r",
66              (pb_sram + i),      rbyte,         wbyte1);
67          return(FAIL);
68       } /* end if */

69       *(pb_sram + i) = wbyte2;   /* write second pattern */
```

```
70        for(j=0; j < delay1; j++);
71        rbyte = *(pb_sram + i);    /* read first time */
72        for(j=0; j < delay1; j++);
73        rbyte = *(pb_sram + i);    /* read second time */
74        PRINTF("%x", rbyte);       /* display the read back byte */
75        if (rbyte != wbyte2)
76        {
77              PRINTF("\n\rLOCATION %xh FAILED!
                                        READ %xh SHOULD READ %xh\n\r",
78                (pb_sram + i), rbyte, wbyte2);
79             return(FAIL);
80        } /* end if */

81     } /* end for */

82        return(PASS);

83    } /* end scpu_5 */
```

scpu_6.c

```
#include   <sys/diagnostic.h>
#include   <sys/firmware.h>
#include   <sys/sbd.h>
#include   <sys/edt.h>
#include   <sys/cio_defs.h>
#include   <ciofw.h>
#include   <iodep.h>
#include   <sys/queue.h>
#include   <phaseload.h>
#include   <per_dgn.h>
#include   <ppc_dgn.h>

#define DEBUG

/**
*
*   Copyright (c) 1986 AT&T
*
*   This routine tests parallel in port of HR1
**/

struct dgnret dgnret;
extern char ph_no;
unsigned short etime;
scpu_6()
{
 register int i, j;
 register int delay1 = 50000;
 long dly1, save_int;
 int pb_slot;     /* slot # of this board */
 int vec_num;     /* interrupt vector number */
 char *pb_id;     /* ID address */
 char *pb_vec;    /* interrupt address */
 char *pb_par;    /* parallel port address */
 char *pb_sero;      /* serial out port address */
 char *pb_seri;      /* serial in port address */
 char byte1;
 char byte2;

 /* phase execution time */

 unsigned short etime = 10;
```

```
38    /* global phase number */

39    ph_no = 6;
40
41    /* print test header */

42    PRINTF("\n\r\nHR1 Phase: %d   Name: SCPU_6   Type: NORMAL\n", ph_no);
43    PRINTF("Test Count: 3   Time: %d sec.\n", etime);

44    /* execute onboard diagnostic */

45    pb_slot = EDTP(OPTION)->opt_slot;  /* get board slot # from edt */

46              /* calculate board access vectors */

47    pb_id = (char *)((pb_slot * 0x200000) + 0x1);  /* ID code reg*/
48    pb_seri = (char *)((pb_slot * 0x200000) + 0x5); /* serial in */
49    pb_vec = (char *)((pb_slot * 0x200000) + 0x7);  /* intvec loc*/
50    pb_sero = (char *)((pb_slot * 0x200000) + 0xfe);/* serial out*/
51    pb_par = (char *)((pb_slot * 0x200000) + 0xff);/*parallel prt*/

52    PRINTF("PARALLEL IN PORT TEST\r\n");
53    PRINTF("PLEASE, START START CHANGING DIP
                                SWITCHES ON MY COMMAND\n\n");
54    for(j=0; i < delay1; j++);
55    for(j=0; i < delay1; j++);
56    for(j=0; i < delay1; j++);
57    for(j=0; i < delay1; j++);
58    PRINTF("GGGGGGO!!!!!\n\n");

59   /* Serial in  test */

60    for(i=0; i < 300; i++)
61       {
62       PRINTF("%x",*pb_par);
63       for(j=0; j < delay1; j++);
64       }

65    return(PASS);

66    } /* end scpu_6 */
```

scpu_7.c

```
 1   #include   <sys/diagnostic.h>
 2   #include   <sys/firmware.h>
 3   #include   <sys/sbd.h>
 4   #include   <sys/edt.h>
 5   #include   <sys/cio_defs.h>
 6   #include   <ciofw.h>
 7   #include   <iodep.h>
 8   #include   <sys/queue.h>
 9   #include   <phaseload.h>
10   #include   <per_dgn.h>
11   #include   <ppc_dgn.h>

12   #define DEBUG

13   /**
14   *
15   *   Copyright (c) 1986 AT&T
16   *
17   **/

18   struct dgnret dgnret;
19   extern char ph_no;
20   unsigned short etime;
21   scpu_7()
22   {
23    register int i, j;
24    register int delay1 = 30000;
25    long dly1, save_int;
26    int pb_slot;      /* slot # of this board */
27    int vec_num;      /* interrupt vector number */
28    char *pb_id;      /* ID address */
29    char *pb_vec;     /* interrupt address */
30    char *pb_par;     /* parallel port address */
31    char *pb_sero;       /* serial out port address */
32    char *pb_seri;       /* serial in port address */
33    char byte1;
34    char byte2;

35    /* phase execution time */

36    unsigned short etime = 2;
```

```
37    /* global phase number */

38    ph_no = 7;
39
40    /* print test header */

41    PRINTF("\n\r\nHR1 Phase: %d, Name: SCPU_7, Type: DEMAND\n", ph_no);
42    PRINTF("Test Count: 3   Time: %d sec.\n", etime);

43    /* execute onboard diagnostic */

44    pb_slot = EDTP(OPTION)->opt_slot;  /* get board slot # from edt */

45              /* calculate board access vectors */

46    pb_id = (char *)((pb_slot * 0x200000) + 0x1);    /* ID code regist.*/
47    pb_seri = (char *)((pb_slot * 0x200000) + 0x5);    /* serial in */
48    pb_vec = (char *)((pb_slot * 0x200000) + 0x7);     /* int vec loc */
49    pb_sero = (char *)((pb_slot * 0x200000) + 0xfe);    /* serial out */
50    pb_par = (char *)((pb_slot * 0x200000) + 0xff);     /* parallel port */

51   /* Start your coding here */

52    return(PASS);

53   } /* end scpu_7 */
```

dummy.c

```
1   main()
2     {
3     /* this is an empty file to satisfy DGMON requirement */
4     }
```

m a k e . h i

```
1   TITLE = High Level makefile for 3B2  -HR1- Diagnostics
2   PRODUCTS = m32 x56
```

iodep.h

```
 1   typedef long   RAPP;
 2   typedef long   CAPP;

 3   #define CQSIZE     10
 4   #define RQSIZE     5
 5   #define NUM_QUEUES 1

 6   #define REQUEST       0   /* request queue */

 7   /*
 8     Number of sub-devices.  The Ports board actually has no
 9     sub devices; however we must make NUM_DEVS at least 1
10     for C declaration purposes.  The initialization value within
11     the subdevice table informs the SBD that there are actually
12     zero devices.
13   */

14   #define NUM_DEVS   1

15   /* Board ID */
16   #define BDID   2
```

per_dgn.h

```
 1   /**
 2   *
 3   *   Common header file for peripheral diagnostics
 4   *   using "phzrun()".
 5   **/

 6    /* PB RAM page size ( K bytes ) */

 7   #define  SEGSIZE     0x100

 8    /* DMA control word ( transfer bytes ) used in phasend() */

 9   #define  DMAB_CW     0xb7ae

10    /* page register value for returning dgn structure */

11   #define  DGN_PAGE 0x0 /* 0x2000000 - 0x201ffff */

12    /* diagnostic return address - this value is used
13    as the destination address for DMA of the diagnostic
14    results to the SBD */

15   #define  DMARETAD 0x8f000

16    /* pointer to diagnostic return structure. this is the
17    only place where the address is actually defined. */

18   #define  DGNRETST ((struct dgnret *)0x200f000)

19    /* character on which to abort diagnostics */

20   #define  ABORTKEY 0x04   /* Control D */

21    /* additional time to allow for phase execution */

22   #define  ETIMEPAD 4  /* seconds(decimal) */

23    /* mode flags for phzrun() - a variable is set to
24    indicate the current process.  If for some reason
25    diagnostics fail, this value can be looked at with
26    a debug monitor to determine what happened and why. */
```

```
27  #define  QUEINIT  0xa11a /* initialization */
28  #define  BSYSGEN  0xa22a /* board sysgen */
29  #define  DOSEXEC  0xa33a /* executing DOS */
30  #define  DWNLOAD  0xa44a /* diagnostic download */
31  #define  DGNEXEC  0xa55a /* executing FCF */
32  #define  DGNRETN  0xa66a /* waiting for dgn results */

33   /* failing return codes for phzrun(), PASS is
34      returned if all is well */

35  #define  RSPERR   0xb11b /* incorrect response */
36  #define  RSPTMOUT 0xb22b /* timeout waiting for response */
37  #define  DGNTMOUT 0xb33b /* timeout during dgn execution */
38  #define  NORESULT 0xb44b /* no diagnostic results returned */
39  #define  UNEXPINT 0xb55b /* unexpected interrupt */
40  #define  UNEXPEXC 0xb66b /* unexpected exception */
41  #define  WRITFAIL 0xb77b /* write of dgn return struct failed */
42  #define  CONABORT 0xbffb /* console interruption */

43   /* diagnostic return structure - if the variable
44   names or types are changed, be sure to update the
45   macros used to reference them */

46  struct dgnret
47  {
48   unsigned short d_flag;  /* pass/fail flag */
49   unsigned short d_ftst;  /* first failing test # */
50   unsigned short d_rawd;  /* raw data */
51   unsigned short d_supd;  /* supplementary data */
52  };

53   /* size of diagnostic return structure ( bytes ) */

54  #define  DGRTSIZE 0x8

55   /* Macros used to access dgnret variables.  The
56   first definition is used by phasend(), the second
57   by presult() */

58   /* pass/fail flag */

59  #define  RESLT    (dgnret.d_flag)
60  #define  PRESLT   (DGNRETST->d_flag)

61   /* failing test # */

62  #define  FFTEST   (dgnret.d_ftst)
```

```
63   #define  PFFTEST (DGNRETST->d_ftst)

64    /* raw data */

65   #define  RAWD (dgnret.d_rawd)
66   #define  PRAWD    (DGNRETST->d_rawd)

67    /* supplementary data */

68   #define  SUPD (dgnret.d_supd)
69   #define  PSUPD    (DGNRETST->d_supd)

70    /* macro used to access completion queue opcode */

71   #define C_opcode(R)
72    ((CQUEUE *)C_ADDR)->queue.entry[R].common.codes.bytes.opcode

73    /* macros used to access express request queue */

74   #define R_Xbytcnt
75    ((RQUEUE *)R_ADDR)->express.common.codes.bytes.bytcnt

76   #define R_Xcmdstat
77    ((RQUEUE *)R_ADDR)->express.common.codes.bits.cmd_stat

78   #define R_Xseqbit
79    ((RQUEUE *)R_ADDR)->express.common.codes.bits.seqbit

80   #define R_Xsubdev
81    ((RQUEUE *)R_ADDR)->express.common.codes.bits.subdev

82   #define R_Xopcode
83    ((RQUEUE *)R_ADDR)->express.common.codes.bytes.opcode

84   #define R_Xaddr
85    ((RQUEUE *)R_ADDR)->express.common.addr

86   #define R_Xappl
87    ((RQUEUE *)R_ADDR)->express.appl.addr

88    /* macro used to access express completion queue opcode */

89   #define C_Xopcode
90    ((CQUEUE *)C_ADDR)->express.common.codes.bytes.opcode
```

phaseload.h

```
 1   /**
 2   *          - phaseload.h -
 3   *
 4   *   This header file defines the load addresses for each
 5   *   x86 diagnostic phase when loaded into SBD RAM.  They
 6   *   are referenced primarily in the ifile "phz_ifile.c".
 7   *
 8   *   These values are also used by the phase startup
 9   *   routine as the source and destination addresses for
10   *   download to the peripheral board and also determine
11   *   the number of bytes to be downloaded.
12   *
13   *   Unfortunately there is no easy way to calculate these
14   *   values.  Each phase was compiled and then it's size
15   *   used to determine starting address and space needed.
16   *
17   *   Utilization of SBD RAM is
18   *
19   *
20   *   0x2000000 -----------------
21   *           ¦ Diagnostic ¦
22   *           ¦ Monitor ¦
23   *   0x200c000 -----------------
24   *           ¦ Diagnostic ¦
25   *           ¦ Phase Table¦
26   *   0x200c??? -----------------
27   *           ¦ SBD      ¦
28   *           ¦ Diagnostic ¦
29   *           ¦ Startup Code   ¦
30   *   0x200???? -----------------
31   *           ¦ SBD Common ¦
32   *           ¦ Diagnostic ¦
33   *           ¦ Routines   ¦
34   *   0x200f000 -----------------
35   *           ¦ Diagnostic ¦
36   *           ¦ Return Struct  ¦
37   *   0x2010100 -----------------
38   *           ¦ Diagnostic ¦
39   *           ¦ Phase      ¦
40   *   0x2011100 -----------------
41   *           ¦ Diagnostic ¦
42   *           ¦ Phase      ¦
```

```
43   *   0x2012100 -----------------
44   *          ¦ Diagnostic  ¦
45   *          ¦ Phase       ¦
46   *          -----------------
47   *          ¦      ¦
48   *          V      V
49   *   0x20d0000 -----------------
50   **/

51   /* define low and high peripheral load addresses */

52   #define  LCSTEST 0x0500 /* used to load low chip select test */
53   #define  LDLORAM 0x1000
54   #define  LDHIRAM 0x5000

55   /* define the starting address for each phase */

56   #define  PHASE01 0x2010100 /* cio */
57   #define  PHASE02 0x2011100 /* pcsr */
58   #define  PHASE03 0x2012100 /* ram_h */
59   #define  PHASE04 0x2013100 /* ram_l */
60   #define  PHASE05 0x2014100 /* rom */
61   #define  PHASE06 0x2015100 /* cpu_1 */
62   #define  PHASE07 0x2016100 /* cpu_2 */
63   #define  PHASE08 0x2017100 /* cpu_3 */
64   #define  PHASE09 0x2018100 /* cpu_4 */
65   #define  PHASE10 0x2019100 /* cpu_5 */
66   #define  PHASE11 0x201a100 /* pio_1 */
67   #define  PHASE12 0x201b100 /* pio_2 */
68   #define  PHASE13 0x201c100 /* DMA byte */
69   #define  PHASE14 0x201d100 /* DMA word */
70   #define  PHASE15 0x201e100 /* print_1 */
71   #define  PHASE16 0x201f100 /* print_2 */
72   #define  PHASE17 0x2020100 /* duart0_1 */
73   #define  PHASE18 0x2022100 /* duart1_1 */
74   #define  PHASE19 0x2024100 /* duart0_2 */
75   #define  PHASE20 0x2025100 /* duart1_2 */
76   #define  PHASE21 0x2026100 /* duart0_3 */
77   #define  PHASE22 0x2027100 /* duart1_3 */
78   #define  PHASEND 0x2028100 /* END OF DIAGNOSTIC PHASES */
```

Appendix C: System Header Files

Contents

Appendix C: System Header Files

The */usr/include/sys* directory and subdirectories includes a number of header files for system data structures and other structures associated with drivers that are bundled with the UNIX operating system. The following sections list the system header files that can be used in driver code.

Hardware-Independent Header Files Used in Drivers

The following header files contain predominantly hardware-independent and implementation-independent information; their contents do not vary substantially between machines or releases. They contain definitions of data structures used to maintain kernel state information, definitions of data objects used throughout the kernel, and the internal flags used as state indicators in the data structures defined here.

buf.h defines the members of the buffer header used with the system buffer cache, including the valid flags for the **b_flags** member. **#include** this header file in all block-access drivers and in character-access drivers that use a buffering scheme that relies on this same header.

cmn_err.h defines the **cmn_err**(D3X) print interface. **#include** in all driver code.

conf.h defines the switch table structures, `bdevsw`(D4X), `cdevsw`(D4X), and `linesw`(D4X).

debug.h defines all facilities available with **cc -DDEBUG**. Drivers that include ASSERT code for debugging should **#include** this file.

elog.h defines external major numbers for use by error logging, statistics used for estimating error rates during error logging, and the structure that tracks I/O activity for system accounting. Drivers for disk, tape, printer, network, and other hardware drivers should **#include** this file.

errno.h defines standard error codes; used in all drivers.

file.h defines the UNIX System V file structure, including valid values for the **f_flag** member; used by drivers that use control flags on **open**(D2X) routine.

immu.h contains the source for the **getsrama**(D3X) and **getsramba**(D3X) macros. *immu.h* is used in memory management.

inline.h redefines the **spl*** functions and contains memory management functions outside the AT&T driver interface.

iobuf.h defines IDFC controller status information and a private buffer header structure for this disk device.

map.h defines the memory mapping scheme discussed in Chapter 6; required for all drivers that use a map to manage dynamically-allocated memory.

open.h defines types of **open**(2) and **close**(2) system calls. These types can be used to determine when these system calls will activate the corresponding driver routines and when they will not. If the device for your driver requires this facility, **#include** this header file and use the defined types as the third argument to the **open**(D2X) and **close**(D2X) routines.

param.h gives parameter definitions that are required by other header files; **#include** after *types.h* in all drivers.

proc.h defines the `proc`(D4X) structure that contains reference to the current process.

signal.h defines **signal** mechanism; required in any driver that uses **signal**(D3X) or **psignal**(D3X).

stream.h defines data structures used for the STREAMS interface; required in any STREAMS-interface driver.

stropts.h defines options and IOCTLs for STREAMS drivers; required in any STREAMS-interface driver.

strstat.h defines the counters used for gathering statistics for the STREAMS interface; required in any STREAMS-interface driver.

sysinfo.h contains several counters and flags used by drivers to record event status, such as when an interrupt routine is serviced.

systm.h defines system entry table, system devices (such as rootdev and swapdev) and system scheduling variables; required for any driver that uses **dma_breakup**(D3X), **drv_rfile**(D3X), **geteblk**(D3X), **logstray**, or **hdelog**(D3X).

termio.h defines the I/O control commands that are supported for terminal drivers; required for all terminal drivers.

trace.h used by the **trace** driver.

tty.h defines structures used for TTY devices, including `clist`(D4X), `ccblock`(D4XX), `cblock`(D4X), `cfreelist`(d4), `tty`(D4X). Also defines commands and flags used with the **tty** line discipline. **#include** in any driver that uses a `cblock` buffering scheme or a TTY structure.

types.h gives type definitions that are required by other header files; **#include** in all drivers, usually before any other header files.

user.h defines the `user`(D4X) structure

vtoc.h defines I/O control commands, error codes, and structures used for VTOC'ed disks. should **#include** in all drivers for VTOC'ed disk devices.

Header Files from Other Drivers

In general, header files defined for one driver should not be used in another driver. The following header files are exceptions

log.h defines the STREAMS log driver, should be included in all STREAMS driver code.

hdelog.h defines drivers structures, tables, and queues used for the Disk Defect Management feature. All drivers for disk devices that run under Disk Defect Management should include this file. See Chapter 11, "Error Reporting," for more information.

strlog.h defines STREAMS log driver interface, should be included in all STREAMS driver code.

System Definition Header Files for I/O

The following UNIX System V header files define the I/O bus of the AT&T 3B2 computer, the common software/firmware, and pumpcode conventions used in all peripherals attached to the system's I/O bus. Also included here are files that describe hardware (such as the DMA controller) used explicitly by more than one device driver. These files may be included by appropriate device drivers.

cio_defs.h defines common status from all I/O applications and drivers and gives macros for common I/O firmware functions.

diskette.h defines diskette formatting structures; required in all drivers for controllers that support diskette devices.

dma.h defines Direct Memory Access (DMA) conventions

io.h defines disk partition tables.

lla.h defines common I/O queue entry opcodes.

pump.h defines pumpcode I/O control commands and other information used when downloading information to an intelligent controller

queue.h defines queue pointer macros.

Appendix D: Sample Character Driver

Contents

Appendix D: Sample Character Driver

Driver Routines

This appendix lists a serial driver that interacts with a Dual Universal Asynchronous Receiver-Transmitter (DUART) such as that used by a terminal.

Table D–1 Driver Routines

Routine	Line Number	Purpose
init	60	initialize variables when system is booted
open	72	start access to device
close	102	complete access to device
read	116	read terminal data
write	124	send character to terminal
ioctl	132	I/O control command routine
int	179	interrupt routine
rint	209	character-received interrupt routine
xint	296	character-transmitted interrupt routine
modem	395	enable/disable modem
param	139	request modem to hang up phone line
proc	318	process input characters

Character Driver Code

```
 1   #include "sys/param.h"
 2   #include "sys/types.h"
 3   #include "sys/signal.h"
 4   #include "sys/dir.h"
 5   #include "sys/immu.h"
 6   #include "sys/psw.h"
 7   #include "sys/pcb.h"
 8   #include "sys/user.h"
 9   #include "sys/errno.h"
10   #include "sys/file.h"
11   #include "sys/tty.h"
12   #include "sys/termio.h"
13   #include "sys/conf.h"
14   #include "sys/sysinfo.h"
15   #include "sys/sysmacros.h"
16   #include "sys/inline.h"

17   struct duart {
18    char   uart_cmnd;    /* command register */
19    char   uart_csr; /* control/status register */
20    char   dtr;      /* data terminal ready status reg */
21    char   dcd;      /* data carrier detect reg*/
22    char   uart_data;    /* receive-transmit data holding reg */
23    char   vector;       /* interrupt vector register */
24    int speed;     /* baud rate register*/
25    int mr1;       /* mode register - channel 1 */
26    int mr2;       /* mode register - channel 2 */
27   };


28   extern struct duart duart[];   /* the uart device */
29   extern struct tty DRVR_tty[];  /* tty data structures */
30   extern int nduart;

31   /*
32    *  Device commands
33    */


34   #define  DISABLE 0
35   #define  ENABLE  1
36   #define  RESET   2
37   #define  STRT_BRK 3
```

```
38  #define  STOP_BRK 4
39  #define  CLEAR_INT   5
40  #define  RESET_ERR   6

41  /*
42   *  Register bits
43   */

44  #define  BITS5    0
45  #define  BITS6    1
46  #define  BITS7    2
47  #define  BITS8    3
48  #define  OPAR     0x10
49  #define  NO_PAR   0x20

50  #define  ONESB    1
51  #define  TWOSB    2

52  #define  RCVRDY   0x01
53  #define  XMTRDY   0x02
54  #define  FE       0x04
55  #define  OVRRUN   0x08
56  #define  PARERR   0x10
57  #define  RCVD_BRK 0x20

58  /* internal major number from master.d file */
59  extern int DRVR_maj;


60  DRVRinit()
61  {
62   int i,j;

63   for(i = 0; i < nduart;i++) {
64       duart[i].uart_cmnd = DISABLE;
65       for (j=0; j<128; j++)
66       if (MAJOR[j] == DRVR_maj && MINOR[j] == i) {
67           duart[i].vector = j << 4;
68           break;
69       }
70   }
71  }


72  DRVRopen(dev, flag)
73  register dev, flag;
74  {
```

```
 75    register struct tty *tp;
 76    int oldpri;
 77    extern DRVRproc();

 78    dev = minor(dev);

 79    if (dev >= nduart) {
 80       u.u_error = ENXIO;
 81       return;
 82    }

 83    tp = &DRVR_tty[dev];
 84    if ((tp->t_state & (ISOPEN | WOPEN)) == 0) {
 85       ttinit(tp);
 86       tp->t_proc = DRVRproc;
 87       DRVRparam(dev);
 88    }

 89    oldpri = spltty();
 90    if (tp->t_cflag & CLOCAL || DRVRmodem(dev, ON))
 91       tp->t_state |= CARR_ON;
 92    else
 93       tp->t_state &= ~CARR_ON;

 94    if ( !(flag & FNDELAY) )
 95       while ((tp->t_state & CARR_ON == 0) {
 96          tp->t_state |= WOPEN;
 97          sleep((caddr_t) & tp->t_canq, TTIPRI);
 98       }

 99    (*linesw[tp->t_line].l_open)(tp);
100    splx(oldpri);
101   }


102   DRVRclose(dev)
103   register dev;
104   {
105    register struct tty *tp;
106    register int  oldpri;

107    dev = minor(dev);

108    tp = &DRVR_tty[dev];
109    (*linesw[tp->t_line].l_close)(tp);

110    if (tp->t_cflag & HUPCL) {
```

```
111         oldpri = spltty();
112         DRVRmodem(dev, OFF);
113         splx(oldpri);
114     }
115     }


116     DRVRread(dev)
117     register dev;
118     {
119     register struct tty *tp;

120     dev = minor(dev);

121     tp = &DRVR_tty[dev];
122     (*linesw[tp->t_line].l_read)(tp);
123     }


124     DRVRwrite(dev)
125     register dev;
126     {
127     register struct tty *tp;

128     dev = minor(dev);

129     tp = &DRVR_tty[dev];
130     (*linesw[tp->t_line].l_write)(tp);
131     }


132     DRVRioctl(dev, cmd, arg, mode)
133     register dev, cmd, arg, mode;
134     {
135     dev = minor(dev);
136     if (ttiocom(&DRVR_tty[dev], cmd, arg, mode))
137         DRVRparam(dev);
138     }


139     DRVRparam(dev)
140     register dev;
141     {
142     register struct tty *tp;
143     register flag, mr1, mr2;
144     int s;
```

```
145    s = spltty();
146    tp = &DRVR_tty[dev];
147    flags = tp->t_cflag;

148    if ((flags & CBAUD) == 0) {
149        /* hang up modem */
150        DRVRmodem(dev, OFF);
151        splx(s);
152        return;
153    }


154    mr1 = 0;
155    if ((flags & CSIZE) == CS8)
156        mr1 |= BITS8;
157    if ((flags & CSIZE) == CS7)
158        mr1 |= BITS7;
159    if ((flags & CSIZE) == CS6)
160        mr1 |= BITS6;
161    if ((flags & PARENB) == 0)
162        mr1 |= NO_PAR;
163    if ((flags & PARODD) != 0)
164        mr1 |= OPAR; /* if not odd, then even assumed */

165    mr2 = 0;
166    if (flags & CSTOPB)
167        mr2 |= TWOSB;
168    else
169        mr2 |= ONESB;

170    (*tp->t_proc)(tp,T_SUSPEND);

171    duart[dev].uart_cmnd = RESET;

172    duart[dev].mr1 = mr1;
173    duart[dev].mr2 = mr2;
174    duart[dev].speed = flags & CBAUD;
175    duart[dev].uart_cmnd = ENABLE;
176    (*tp->t_proc)(tp,T_RESUME);

177    splx(s);
178   }


179   DRVRint(dev)
180   register dev;
181   {
```

```
182    register struct tty *tp;
183    register char sr;

184    dev = 0;
185    tp = &DRVR_tty[dev];

186    duart[dev].uart_cmnd = CLEAR_INT;
187    if (tp->t_cflag & CLOCAL || duart[dev].dcd) {
188        if ((tp->t_state & CARR_ON) == 0) {
189            wakeup(&tp->t_canq);
190            tp->t_state |= CARR_ON;
191        }
192    } else {
193        if (tp->t_state & CARR_ON) {
194            if (tp->t_state & ISOPEN) {
195                signal(tp->t_pgrp, SIGHUP);
196                duart[dev].dtr = OFF;
197                ttyflush(tp, (FREAD | FWRITE));
198            }
199            tp->t_state &= ~CARR_ON;
200        }
201    }

202    /* check status register */
203    sr = duart[dev].uart_csr;

204    if (sr & RCVRDY)
205        DRVRrint(dev);
206    if (sr & XMTRDY)
207        DRVRxint(dev);
208    }


209    DRVRrint(dev)
210    register dev;
211    {
212     register struct tty *tp;
213     register char c, stat;
214     register char *sr;
215     register struct ccblock *rbuf;

216     sysinfo.rcvint++;
217     if (dev >= nduart)
218        return;
219     tp = &DRVR_tty[dev];

220     sr = &duart[dev].uart_csr;
```

```
221    while ((stat = *sr) & RCVRDY) {
222        c = duart[dev].uart_data;

223        /* check for CSTART/CSTOP */

224        if (tp->t_iflag & IXON) {
225            register char ctmp;
226            ctmp = c & 0177;
227            if (tp->t_state & TTSTOP) {
228                if (ctmp == CSTART || tp->t_iflag & IXANY)
229                    (*tp->t_proc)(tp, T_RESUME);
230            } else {
231                if (ctmp == CSTOP)
232                    (*tp->t_proc)(tp, T_SUSPEND);
233            }
234            if (ctmp == CSTART || ctmp == CSTOP)
235                continue;
236        }


237        /* Check for errors */
238        {
239            register int flg;
240            char lbuf[3];/* local character buffer */
241            short lcnt;  /* count of chars in lbuf */

242            lcnt = 1;
243            flg = tp->t_iflag;
244            if (stat & (FE | PARERR | OVRRUN))
245                duart[dev].uart_cmnd = RESET_ERR;

246            if (stat & PARERR && !(flg & INPCK))
247                stat &= ~PARERR;

248            if (stat & (RCVD_BRK | FE | PARERR | OVRRUN)) {
249                if ((c & 0377) == 0) {
250                    if (flg & IGNBRK)
251                        continue;

252                    if (flg & BRKINT) {
253                        (*linesw[tp->t_line].l_input)(tp,
254                            L_BREAK);
255                        continue;
256                    }
```

```
257             } else {
258                 if (flg & IGNPAR)
259                     continue;
260             }

261             if (flg & PARMRK) {
262                 lbuf[2] = 0377;
263                 lbuf[1] = 0;
264                 lcnt = 3;
265                 sysinfo.rawch += 2;
266             } else
267                 c = 0;
268         } else {
269             if (flg & ISTRIP)
270                 c &= 0177;
271             else {
272                 c &= 0377;
273                 if (c == 0377 && flg & PARMRK) {
274                     lbuf[1] = 0377;
275                     lcnt = 2;
276                 }
277             }
278         }

279         lbuf[0] = c;
280         rbuf = &tp->t_rbuf;
281         while (lcnt) {
282             *rbuf->c_ptr++ = lbuf[--lcnt];
283             if (--rbuf->c_count == 0) {
284                 rbuf->c_ptr -= rbuf->c_size;
285                 (*linesw[tp->t_line].l_input)(tp,
286                     L_BUF);
287             }
288         }
289         if (rbuf->c_size != rbuf->c_count) {
290             rbuf->c_ptr -= rbuf->c_size - rbuf->c_count;
291             (*linesw[tp->t_line].l_input)(tp, L_BUF);
292         }
293     }
294  }
295 }


296 DRVRxint(dev)
297 register dev;
298 {
299  register struct tty *tp;
```

```
300    register char *sr;

301    sysinfo.xmtint++;
302    tp = &DRVR_tty[dev];
303    if (tp->t_state & TTXON) {
304        tp->t_state |= BUSY;
305        duart[dev].uart_data = CSTART;
306        tp->t_state &= ~TTXON;
307    } else
308    if (tp->t_state & TTXOFF) {
309        tp->t_state |= BUSY
310        duart[dev].uart_data = CSTOP;
311        tp->t_state &= ~TTXOFF;
312    } else
313    if (tp->t_state & BUSY && !(tp->t_state&(TIMEOUT|TTSTOP))) {
314        tp->t_state &= ~BUSY;
315        DRVRproc(tp, T_OUTPUT);
316    }
317   }


318   DRVRproc(tp, cmd)
319   register struct tty *tp;
320   register cmd;
321   {
322    register dev;
323    int s;
324    extern ttrstrt();

325    s = spltty();
326    dev = tp - DRVR_tty;
327    switch(cmd) {
328    case T_TIME:
329        if (tp->t_state&TIMEOUT) {
330            tp->t_state &= ~TIMEOUT;
331            duart[dev].uart_cmnd = STOP_BRK;
332        }
333        goto start;

334    case T_WFLUSH:
335        tp->t_tbuf.c_size -= tp->t_tbuf.c_count;
336        tp->t_tbuf.c_count = 0;

337    case T_RESUME:
338        tp->t_state &= ~TTSTOP;
339        goto start;
```

```
340    case T_OUTPUT:
341        start:
342        {
343            register struct ccblock *tbuf;

344            if (tp->t_state & (BUSY | TTSTOP | TIMEOUT))
345                break;
346            tbuf = &tp->t_tbuf;

347            /* check if tbuf is empty */
348            if (tbuf->c_ptr == NULL || tbuf->c_count == 0) {
349                if (tbuf->c_ptr)
350                    tbuf->c_ptr -= tbuf->c_size;
351            if (!(CPRES&(*linesw(tp->t_line].l_output)(tp)))
352                    break;
353            }
354            tp->t_state |= BUSY;
355            duart[dev].uart_data = *tbuf->c_ptr++;
356            tbuf->c_count--;
357            break;
358        }


359    case T_SUSPEND:
360        tp->t_state |= TTSTOP;
361        break;

362    case T_BLOCK:
363        tp->t_state &= ~TTXON;
364        tp->t_state |= TBLOCK;

365        if (tp->t_state & BUSY)
366            tp->t_state |= TTXOFF;
367        else {
368            tp->t_state |= BUSY;
369            duart[dev].uart_data = CSTOP;
370        }
371        break;

372    case T_RFLUSH:
373        if (!(tp->t_state & TBLOCK))
374            break;

375    case T_UNBLOCK:
376        tp->t_state &= ~(TTXOFF | TBLOCK);

377        if (tp->t_state & BUSY)
```

```
378             tp->t_state |= TTXON;
379         else {
380             tp->t_state |= BUSY;
381             duart[dev].uart_data = CSTART;
382         }
383         break;


384     case T_BREAK:
385         duart[dev].uart_cmnd = STRT_BRK;
386         tp->t_state |= TIMEOUT;
387         timeout(ttrstrt, tp, HZ/4);
388         break;

389     case T_PARM:
390         DRVRparam(dev);
391         break;
392     }   /* end of switch cmd */
393     splx(s);
394   }


395   DRVRmodem(dev, flag)
396   register dev, flag;
397   {
398     register bit;

399     if (flag == OFF)
400         duart[dev].dtr = OFF;
401     else
402         duart[dev].dtr = ON;

403     return( duart[dev].dcd );
404   }
```

Appendix E: Sample Block Driver

Contents

Appendix E: Sample Block Driver

The **doc_** driver is a block driver for a disk controller that runs on the Single Board Computer (SBC). This driver is an example of a working hardware driver for a block-access device that also supports character access.

Table E-1 summarizes the driver entry point routines (*BCI Driver Reference Manual,* Section D3X), kernel functions used in each, and the subordinate routines each calls. The initial line number of each routine is given in parentheses following the routine name.

Table E−1 doc_ Driver Routine Summary

Entry Table	Entry Point Routine Name	Subordinate Routines
`io_init`	**doc_init**	**doc_initdr** **doc_gocheck**
`bdevsw` or `cdevsw`	**doc_open**	**doc_copy, doc_setblk, doc_strategy**
	doc_close	
`bdevsw`	**doc_strategy**	**doc_iostart**
`cdevsw`	**doc_read**	**doc_breakup, doc_strategy**
	doc_write	**doc_breakup, doc_strategy**
	doc_ioctl	
Interrupt Vector Table	**doc_int**	**doc_intr, doc_iostart**

This appendix includes the full master file and header file for the driver in addition to the full driver code. The lines in the driver code are numbered sequentially, with section headers inserted for ease of reference. Note that some lines had to be split to fit on the physical page. The continuation portions of such lines are not given numbers.

doc_ Driver Master File

The values assigned to the first six columns of the master file indicate the following:

FLAG This driver supports both block and character access.

VEC Each device controlled by this driver has one interrupt vector. This indicates that the device itself must have some way of indicating which subdevice generated an interrupt, which is typical of intelligent disk controllers. Because the value of #VEC is not double the number in #DEV, **lboot** will create an entry for the **doc_int** routine in the Interrupt Vector Table rather than **doc_rint** and **doc_xint** entries.

PREFIX The prefix for this driver is "doc_", so the entry point routines will be named "doc_open," "doc_close," and so forth.

SOFT This field has no number in it, so this is not a software driver; the external major number for devices controlled by this driver is determined by the board slot of the device, not the master file.

#DEV Each **doc_** device (controller) can support a maximum of four subdevices.

IPL Devices controlled by this driver will interrupt at priority level 10, which is the appropriate IPL for a disk device. Checking the table on the **spl***n*(D3X) reference page, you see that, on the SBC, this means that critical code protected by **spl5** or higher will not be interrupted by devices controlled by this driver.

```
1   *-------------------------------------------------------------------
2   * Master file for doc_ disk controller.
3   *
4   * DOC_
5   *
6   * NOTE: doc_cpaddr is array, maximum [#C] size
7   *    (set by initializer below)
8   *
9   *-------------------------------------------------------------------
10  *FLAG   #VEC    PREFIX   SOFT    #DEV    IPL  DEPENDENCIES/VARIABLES
11  bc      1       doc_     -       4       10
12  *-------------------------------------------------------------------
13  * Controller physical addresses.
14  * These are VME A24 physical addresses.
15                                          doc_cpaddr (%i%i) = {
16                                                      0xfd0000,
17                                                      0xfe0000
18                                          }
19  *-------------------------------------------------------------------
20  * Drive types.
21  * Floppy disk drive is drive select 0.
22  * 1st hard disk drive is usually drive select 2, because that is
23  * what installation scripts (on installation floppies) mandate.
24  * 2nd hard disk drive should be set at drive select 1, because
25  * there is some hardware funniness about drive select 3. The
26  * funniness is that whenever no drive is being accessed, drive 3
27  * gets selected.  Upon power-up or power-down, drive 3 is selected
28  * but the control lines may glitch as power ramps up or down.  So
29  * there may be a risk of corruption of the drive set to drive
30  * select 3.
31                                          doc_itype (%i%i%i%i) = {
32                                                    FLOPPY,
33                                                    HARD,
34                                                    HARD,
35                                                    HARD
36                                          }
37  *-------------------------------------------------------------------
38  * Driver internal major number.
39                                          doc_intmaj (%i) = {#M}
40  *-------------------------------------------------------------------
```

Figure E–1 doc_ Master File (*part 1 of 2*)

```
41   * Controller virtual addresses.
42                                                  doc_caddr[#C] (%i)
43   *-------------------------------------------------------------------
44   * VTOCs.
45                                                  doc_vtoc[#C*#D] (%0x108)
46   *-------------------------------------------------------------------
47   * Drive types.
48                                                  doc_type[#C*#D] (%s)
49   *-------------------------------------------------------------------
50                                                  doc_tab[#C] (%0x44)
51                                                  doc_iostat[#C*#D] (%0x10)
52                                                  doc_count[#C*#D] (%i)
53                                                  doc_tcount[#C*#D] (%i)
54                                                  doc_time[#C*#D] (%0x20)
55                                                  doc_info[#C*#D] (%i)
56                                                  doc_fmtflag[#C] (%i)
57                                                  doc_retrys[#C*#D] (%c)
58                                                  doc_defect[#C*#D] (%0x800)
59                                                  doc_elog[#C*#D] (%0x20)
60                                                  doc_pdsect[#C*#D] (%0x200)
61                                                  doc_tbufon[#C*#D] (%i)
62   *-------------------------------------------------------------------
63   * Number of equipped controllers.
64                                                  doc_numcontr (%i) = {#C}
65   *-------------------------------------------------------------------
66   $$$

67   *    Drive Types
68   HARD =                      0
69   FLOPPY                      =                          1
70   STREAM                      =                          2
71   NODRIVE                     =                          3
72   *-------------------------------------------------------------------
```

Figure E–1 doc_ Master File *(part 2 of 2)*

The DEPENDENCIES/VARIABLES column defines a number of variables that are declared and used in the driver. Note how this master file includes comments that explain what these variables are. The Table E-2 shows the line numbers from the driver code where each of these variables are declared and used.

Table E–2 DEPENDENCIES/VARIABLES Declarations

Variable Name	Declared on Line Number	Used on Line Number(s)
doc_cpaddr	217	303
doc_itype	159	331
doc_intmaj	317	317, 409
doc_caddr	33	35 – 54, 218, 302, 306
doc_vtoc	146	496, 640, 703, 704, 790, 796, 802, 803, 955, 1153, 1197, 1263, 1348, 1357, 1621
doc_type	160	22, 333, 336, 341, 346, 351, 356, 361, 366, 370, 489, 534, 562, 575, 638, 998, 1037,1142, 1144, 1162, 1230, 1234, 1502, 1510, 1603, 1604, 1610, 1620, 1638, 1642, 1697, 1757, 1761
doc_tab	175	329, 330, 809, 919, 1100, 1475
doc_iostat	176	330
doc_count	180	975, 993, 1042, 1297
doc_tcount	184	983, 1296, 1297
doc_time	188	840, 841, 865, 1315, 1316
doc_info	192	327, 497, 524, 554, 610, 618, 619, 624, 625, 639, 641, 642, 711, 724, 728, 729, 774
doc_fmtflag	202	414, 930, 1106, 1109, 1110, 1117, 1140, 1188, 1220, 1222, 1223, 1487, 1490, 1513, 1515, 1520
doc_retrys	222	328, 1141, 1143, 1145, 1161, 1189, 1190, 1205, 1322
doc_defect	226	631, 632, 689, 948, 1613
doc_elog	230	1243, 1253, 1255, 1262, 1266, 1274, 1275, 1277
doc_pdsect	236	369, 494, 571-574, 626, 781, 949, 1154, 1198, 1264, 1281, 1301, 1435, 1708-1733
doc_tbufon	140	326, 1038, 1041, 1157, 1201

doc_ Driver Header File

The header file for the **doc_** driver defines a number of structures and variables that are used in the driver and board registers with which the driver must interact. By defining structures and variables in the header file rather than the driver code itself, you make the driver easier to read and maintain because all related information is listed together. When modifying the driver to run on a different machine or for an updated version of the hardware, you can modify the header file rather than recode the driver.

```
1   /*-----------------------------------------------------------------
2    *              DOC_ disk controller include file
3    *-----------------------------------------------------------------
4    */
5   #define u_short unsigned short
6   #define u_char  unsigned char
7   #define u_long  unsigned long

8   /*
9    * custom ioctl calls: set these so they don't conflict with vtoc.h
10   * ioctl defs DTRACE is func entry, exit and progress points DPRINT
11   * is selected info prints
12   */
13  #define IOCTL_DTRACEOFF 0x0100
14  #define IOCTL_DTRACEON  0x0101
15  #define IOCTL_DPRINTOFF 0x0110
16  #define IOCTL_DPRINTON  0x0111

17  /*
18   *    per disk type control structure  (used by firmware only)
19   */
20  struct doc_types {
21          int mt_maxbn;           /* largest block number (calculated) */
22          int mt_ncyl;            /* number of cylinders */
23          int mt_nhead;           /* number of tracks per cylinder */
24          int mt_nsectrk;  /* number of sectors per track */
25          int mt_seclen;          /* sector length (bytes) */
26  };
```

Figure E–2 doc_.h Header File *(part 1 of 4)*

```
27  /*-------------------------------------------------------*/
28  /*
29   *    variables for accessing DPDRAM
30   */

31  extern unsigned int doc_caddr[];  /* base addrs of cntrllers */

32  /* same for all commands */
33  #define DOC_GOFLAG(C)    (*((unsigned char *)(doc_caddr[C]+0x1)))
34  #define DOC_COMMAND(C)   (*((unsigned short *)(doc_caddr[C]+0x2)))
35  #define DOC_ERRCODE(C)   (*((unsigned char *)(doc_caddr[C]+0x5)))
36  #define DOC_DRIVENO(C)   (*((unsigned char *)(doc_caddr[C]+0x7)))
37  #define DOC_IVECTOR(C)   (*((unsigned char *)(doc_caddr[C]+0x11)))

38  /* for "init drive" command */
39  #define DOC_NHEADS(C)    (*((unsigned char *)(doc_caddr[C]+0x9)))
40  #define DOC_MAXCYL(C)    (*((unsigned short *)(doc_caddr[C]+0xA)))
41  #define DOC_NSECTRK(C)   (*((unsigned char *)(doc_caddr[C]+0xD)))
42  #define DOC_NBYTSEC(C)   (*((unsigned short *)(doc_caddr[C]+0xE)))
43  #define DOC_HDGAP(C)     (*((unsigned char *)(doc_caddr[C]+0x13)))

44  /* for "initialize track buffer" command */
45  #define DOC_TBADDR_H(C)  (*((unsigned short *)(doc_caddr[C]+0xC)))
46  #define DOC_TBADDR_L(C)  (*((unsigned short *)(doc_caddr[C]+0xE)))

47  /* for "force read/write", "read/write with buffering" and "write
48        with track buffer and verify" commands */
49  #define DOC_LBN_H(C)     (*((unsigned short *)(doc_caddr[C]+0x8)))
50  #define DOC_LBN_L(C)     (*((unsigned short *)(doc_caddr[C]+0xA)))
51  #define DOC_SBADDR_H(C)  (*((unsigned short *)(doc_caddr[C]+0xC)))
52  #define DOC_SBADDR_L(C)  (*((unsigned short *)(doc_caddr[C]+0xE)))

53  /*-------------------------------------------------------------*/
54  /* command and status value definitions */

55  /* "go flag" definitions */
56  #define GO_DONE              0x00
57  #define GO_START  0x01
```

Figure E–2 doc_.h Header File *(part 2 of 4)*

```
58   /* command word bit definitions */
59   #define CMD_READ                0x0001
60   #define CMD_WRITE               0x0000
61   #define CMD_VERIFY              0x0002
62   #define CMD_FORCE               0x0004
63   #define CMD_INTWD               0x0008
64   #define CMD_INITTB              0x0010
65   #define CMD_INITDR              0x0020
66   #define CMD_FORMAT              0x0040
67   #define CMD_DMAIO               0x0080
68   #define CMD_FLIO                0x0100
69   #define CMD_HDIO                0x0200
70   #define CMD_STATUS              0x0400
71   #define CMD_FLCMD               0x0800
72   #define CMD_DDENC               0x0000
73   #define CMD_SDENC               0x1000
74   #define CMD_ENBAUTOFL           0x2000
75   #define CMD_DISAUTOFL           0x2001
76   #define CMD_STARTS0             0x4000
77   #define CMD_RESERVED            0x8000
78   /* command word complete commands */
79   #define CMD_RESET               0x4242
80   /* "error register" definitions */
81   #define ERR_NOERROR             0x00
82   #define ERR_DNOTREADY           0x81
83   #define ERR_RESERVED            0x82
84   #define ERR_ACCESSERR           0x83
85   #define ERR_VERIFYERR           0x84
86   #define ERR_DMAERR              0x85
87   #define ERR_DRVNOTINIT          0x86
88   #define ERR_NUMTBS              0x87
89   #define ERR_ILLEGALCMD          0x88
90   #define ERR_ILLEGALLBN          0x89
91   #define ERR_CRCERR              0x8A
92   #define ERR_SEEKERR             0x8B
93   #define ERR_WRITEPROT           0x8C
94   #define ERR_BADMEDIA            0x8D
```

Figure E–2 doc_.h Header File *(part 3 of 4)*

```
 95   /*-----------------------------------------------------------------*/
 96    * addresses of on-board track buffers */ hard disk track == 9K
 97    * bytes, floppy disk track == 4.5K bytes. total internal 7400
 98    * memory for track buffers == 24K bytes (at present).
 99    */
100   #define BUFRAMBASE      0x00022000
101   #define TBADDR_H0       BUFRAMBASE
102   #define TBADDR_H1       TBADDR_H0+0x2400
103   #define TBADDR_F0       TBADDR_H1+0x2400

104   /* track buffer addresses (code assumes
105    * that these go hard, hard, floppy)
106    */
107   static unsigned int tbaddr[3] = { TBADDR_H0, TBADDR_H1, TBADDR_F0 };
108   #define NTB    3

109   /*-----------------------------------------------------------------*/
110   /* hard disk gap parameters */
111   /* these are the numbers given to the controller;
112    * the actual gap is this number plus 3.
113    */
114   #define HDG_256         19
115   #define HDG_512         16

116   /*-----------------------------------------------------------------*/
117   /* defines for splitting int into shorts */
118   #define hihalf(X)       ((short)((X)>>16))
119   #define lohalf(X)       ((short)((X)&0x0000FFFF))

120   /*-----------------------------------------------------------------*/
```

Figure E–2 doc_.h Header File (*part 4 of 4*)

Initial Comment Block

The initial comment block for the **doc_** driver includes a log of all modifications made to the driver and other miscellaneous information that will ease maintenance of the driver. Note that each change that is logged is accompanied with a date.

Line 102 is the control information used by the S-list capability of the C programming language utilities.

```
 1    * revision history:
 2    *
 3    *  051587  DOC_
 4    *
 5    *  - changed to have hard disks be drives 1,2,3 instead of 2,3.
 6    *  - changed doc_diskmaj to doc_intmaj.
 7    *  - changed majnum to extmaj.
 8    *
 9    *  022787  DOC_  1.4
10    *
11    *  - moved "majnum" calculation in doc_init earlier and replaced
12    *    "DOC_0" with "majnum".  Calculation of "majnum" will not
13    *    work correctly for multiple controllers.
14    *  - removed "+1" in doc_iostart calculation of firstbn, and "-1"
15    *    in doc_int error message printing.  Defect table always assumes
16    *    sectors start at 0 now.
17    *  - changed doc_int hard-disk error logging so correct block number
18    *    is used and message is printed before hdelog is called.  Case
19    *    where a bad sector is mapped to a "good" one and the "good" one
20    *    causes an error will still not work.  The original bad sector
21    *    will be logged instead of the "good" one.  Corrected messages
22    *    so proper distinction is made between logical and physical
23    *    accesses.
24    *  - removed "not full disk" message from doc_open.
25    *  - removed hard-coded "hard_pdsect"; replaced with just enough to
26    *    read real pdsect.  This required that the "init drive" code be
27    *    moved out to a new function, doc_initdr, and called
28    *    in a couple of places.
```

Figure E–3 Revision History *(part 1 of 3)*

```
29    *  - these changes should make everything but multiple controllers
30    *    work in SVR3.1.
31    *  - comments need improving; note "majnum" is external major
32    *    number, "doc_diskmaj" is internal major number, and so on.
33    *
34    *  111386  DOC_  1.3
35    *
36    *  - added goflag check before initial reset in case it was
37    *    busy from firmware driver hand-off during boot.
38    *
39    *
40    *  092986  DOC_  1.2
41    *
42    *  - improve error detection for cases where DOC_ board does not
43    *    respond within 1 second after starting a command.  After
44    *    unusually long failures to perform some operation on a drive,
45    *    action should be to stop the requested operation rather than
46    *    continue as did the original driver.
47    *  - add timeout test BEFORE ALL controller commands if go-flag
48    *    wasn't clear; original driver just reported the unclear
49    *    go-flag and continued, now it will wait about 1 sec then
50    *    exit with a message.
51    *  - do the same thing AFTER ALL NON-INTERRUPT-SETTING commands;
52    *    original driver did a wait forever, now it will wait for 1
53    *    second and exit with a message.
54    *
55    *
56    *  082986  DOC_  1.1
57    *
58    *  fix 9 head problem, misc. cleanups:
59    *  - open: set OPEN flag if fulldisk on badopen to avoid the
60    *    sanity reload chicken/egg problem.
61    *  - ioctl PDSETUP: removed "generic values" test.
62    *  - struct hard_pdsect: changed dflt to 9 head disk defaults
63    *    (prob not nec, but just as well changed).
64    *  - cpaddr: moved values to master.d file instead of being
65    *    hard coded (users need reconfig flexibility).
```

Figure E–3 Revision History (*part 2 of 3*)

```
 66    *  - ioctl: added cmds to turn on/off the debug prints so
 67    *    recompile isn't necessary to change it; added TRACE.
 68    *  - while in the code, cleaned up a few minor things in
 69    *    printing messages, shortened messages so the console
 70    *    terminal doesn't lose so much output, removed some unused
 71    *    variables, added a few messages for end-cases, and so on.
 72    *  - errors: changed logic to force single-sector reads or writes
 73    *    after disk errs (code 83 on hards, codes 8A and 8B on
 74    *    floppies); for hard disks, this allows flaw mapping to be at
 75    *    the sector level instead of the track level, so hde error
 76    *    logging, and so on, works; before, it overflowed reloc-sector
 77    *    tables, hdelog, and so on.  When there were many
 78    *    manufacturer's defects (the normal case).  NOTE: the
 79    *    formatdisk flaw entry "T" option is no longer necessary for
 80    *    the DOC_; includes extern doc_tbufon in master.d.
 81    *  - extern variable ndoc_ violated kernel rules for naming
 82    *    globals, changed to doc_numcontr.
 83    *  - a block number calc in doc_intr was using a short which
 84    *    gave a bad block number--changed to an int.
 85    *  - biased blk number by + 1 before sending to hde so hdefix -a
 86    *    works correctly; it still reports wrong but does map the
 87    *    c-t-s in the same way as formatdisk preentry does it (s+=1)
 88    *    so they are consistent;
 89    *
 90    * Original notes on DOC_:
 91    *
 92    * Note: DOC_ only seems to work for disks with 8 heads or less,
 93    *    may not work with "their" disks, and the "get status"
 94    *    command may not work correctly.
 95    *
 96    * Note: This driver does not support cartridge tape.
 97    *
 98    * Note: Since the DOC_ does track buffering, defects must
 99    *       be entered with the "T" option (bad track) under
100    *       formatdisk.
101    *-----------------------------------------------------------------*/

102   #ident          "@(#)kern:doc.c         1.4"
```

Figure E–3 Revision History *(part 3 of 3)*

Global Data Structure Declarations

The driver code itself begins by declaring and defining a number of global data structures that will be used throughout the code. First system and driver-specific header files are **#included**, then the structures defined in the master file and other structures are declared. A number of structures are defined here that could have been defined in the header file. Note how virtually every structure declared or defined is given at least a brief comment that explains its purpose.

```
103   #include "sys/types.h"
104   #include "sys/param.h"
105   #include "sys/sbd.h"
106   #include "sys/vtoc.h"
107   #include "sys/doc_.h"
108   #include "sys/dma.h"
109   #include "sys/immu.h"
110   #include "sys/dir.h"
111   #include "sys/sysmacros.h"
112   #include "sys/signal.h"
113   #include "sys/psw.h"
114   #include "sys/pcb.h"
115   #include "sys/user.h"
116   #include "sys/errno.h"
117   #include "sys/buf.h"
118   #include "sys/elog.h"
119   #include "sys/iobuf.h"
120   #include "sys/systm.h"
121   #include "sys/firmware.h"
122   #include "sys/cmn_err.h"
123   #include "sys/hdelog.h"
124   #include "sys/open.h"
125   #include "sys/inline.h"
126   #include "sys/if.h"
```

Figure E–4 doc_ Global Data Structure Declarations (*page 1 of 6*)

```
127   #define GOWAITSECS 1 /* max time to wait for cntrlr to clr go flag */
128   #define GOCHECKLPS 300000    /* loops, make it come out to seconds */

129   int doc_dtrace = 0; /* debug prints at start, rtrn & go thru funcs */
130   int doc_dprint = 0; /* specific debug prints */

131   #define DTRACE          if(doc_dtrace)printf
132   #define DPRINT          if(doc_dprint)printf
133   #define DEBUGinit       if(doc_dprint)printf
134   #define DEBUGform       if(doc_dprint)printf
135   #define DEBUGnums       if(doc_dprint)printf
136   #define DEBUGdefect     if(doc_dprint)printf
137   #define DEBUGretry      if(doc_dprint)printf
138   #define DEBUGhde        if(doc_dprint)printf

139   #define TBUFFER 1  /* 1 for track buffering, 0 otherwise */
140   extern int doc_tbufon[] ;

141   extern int doc_numcontr; /* num of doc_00 cntrlrs in master file*/

142   #define HRETRYS  5  /* num of positioning retrys for hard disks */
143   #define FRETRYS  1  /* num of positioning retrys for floppy disks*/

144   #define DOC_FRSTBLK  0
145   #define DOC_NULL     0

146   extern struct vtoc doc_vtoc[];        /* in core copy of vtoc */

147   /*  doc_type is set in the master file (i.e. master.d/doc_)
148    *  to reflect the type of disks connected to the controller.
149    *  Each element in doc_type corresponds to the unit number
150    *  of the controller
151    */

152   struct doc_t {
153       int unit0;
154       int unit1;
155       int unit2;
156       int unit3;
157   };
```

Figure E−4 doc_ Global Data Structure Declarations (*page 2 of 6*)

```
158   extern struct doc_t doc_itype;

159   int *doc_itype = (int *)&doc_itype;

160   extern short doc_type[];

161   /*
162    *    Possible types of disk
163    */

164   #define DT_HARD         0
165   #define DT_FLOPPY       1
166   #define DT_STREAMING    2
167   #define DT_NODRIVE      3

168   /* given a unit num (0-(4*C-1)), return controller num (0-(C-1))*/
169   #define contr(x) ((x)>>2)
170   /* given a unit num (0-(4*C-1)), return subdevice number (0-3) */
171   #define subdev(x) ((x)&0x3)

172   /*
173    *    the io queue headers
174    */

175   extern struct iobuf doc_tab[];
176   extern struct iostat doc_iostat[];        /* errlog */

177   /*
178    *    total count of amount of data transferred so far
179    */

180   extern int doc_count[];

181   /*
182    *    the size of the current io being done on this unit
183    */
```

Figure E–4 doc_ Global Data Structure Declarations (*page 3 of 6*)

```
184   extern int doc_tcount[];

185   /*
186    *     IO performance stats area
187    */

188   extern struct iotime    doc_time[];

189   /*  These are used to give us current
190    *  information about the drive
191    */

192   extern int doc_info[];

193   #define     INFO_NULL       0x00    /* uninitialized */
194   #define     INFO_EQUIPPED   0x01    /* drive equipped */
195   #define     INFO_OPEN       0x02    /* open complete */
196   #define     INFO_OPENING    0x04    /* open not yet complete */

197   /*    flags used during formatting :
198    *
199    *    FMT_IDLE == no format in progress on that controller
200    *    FMT_INPROGRESS == format in progress
201    *    FMT_SUCCEED == format finished and succeeded but
202    *                   IOCTL not awake
203    *    FMT_FAIL == format finished & failed but IOCTL not awake
204    */
205   extern int doc_fmtflag[];
206   #define FMT_IDLE    0
207   #define FMT_INPROGRESS    1
208   #define FMT_SUCCEED    2
209   #define FMT_FAIL    3

210   /*
211    *     physical VME addresses of controller boards;
212    *         the order decides the unit numbers.
213    *         this will be determined from the EDT.
214    *     The VIRTUAL addresses will be calculated by sptalloc
215    *         and stored in doc_caddr[].
216    */
```

Figure E–4 doc_ Global Data Structure Declarations (*page 4 of 6*)

```
217   extern unsigned int doc_cpaddr[];         /* physical */
218   extern unsigned int doc_caddr[];          /* virtual */

219   /*
220    *    retry count for positioning errors
221    */

222   extern char doc_retrys[];

223   /*
224    *    disk defect maps
225    */

226   extern struct defstruct doc_defect[];
227   /*
228    *     Error logging structures
229    */

230   extern struct hdedata doc_elog[];
231   extern hdelog();
232   static int doc_initdr();

233   /*    Physical information from Physical Descriptor
234    *       sector (block 0)
235    */

236   extern struct pdsector doc_pdsect[];

237   /*
238    *     Physical Descriptor information for initializing
239    *        pdsect on floppy drives
240    */
241   #define IFNUMSECT    9
242   #define IFBYTESCT    512
243   #define IFPDBLKNO    1422
```

Figure E–4 doc_ Global Data Structure Declarations (*page 5 of 6*)

```
244   static struct pdinfo floppy_pdsect = {
245       1,                              /* driveid */
246       VALID_PD,                       /* sanity */
247       1,                              /* version */
248       "",                             /* serial */
249       IFTRKSIDE,                      /* cyls */
250       IFNTRAC,                        /* tracks */
251       IFNUMSECT,                      /* sectors */
252       IFBYTESCT,                      /* bytes */
253       0,                              /* logicalst */
254       IFTRACKS * IFNUMSECT - 1,       /* errlogst */
255       IFBYTESCT,                      /* errlogsz */
256       0xffffffff,                     /* mfgst */
257       0xffffffff,                     /* mfgsz */
258       IFPDBLKNO + 1,                  /* defectst */
259       IFBYTESCT,                      /* defectsz */
260       1,                              /* relno */
261       IFPDBLKNO + 2,                  /* relst */
262       IFNUMSECT * 2 - 3,              /* relsz */
263       IFPDBLKNO +2                    /* relnext */
264   };
265   /*
266    *    partition information for floppy disks
267    */
268   static struct partition floppy_sizes[IF_NUMPAR] = {
269       0, 0, 432, 990,          /* partition 0 - cyl 24-78 */
270       0, 0, 612, 810,          /* partition 1 - cyl 34-78 */
271       0, 0, 810, 612,          /* partition 2 - cyl 45-78 */
272       0, 0, 1008, 414,         /* partition 3 - cyl 56-78 */
273       0, 0, 1206, 216,         /* partition 4 - cyl 67-78 */
274       V_ROOT, 0, 18, 1404,     /* partition 5 - cyl 1-78  */
275       V_BACKUP, 0, 0, 1422,    /* partition 6 - cyl 0-78  */
276       V_BOOT, 0, 0, 18         /* partition 7 - cyl 0     */
277   } ;
278   /*
279    *    Misc stuff for decoding device numbers
280    */
281   #define doc_hard(p) (subdev(p) != 0) /* units 1,2,3=hard disks */
282   extern int doc_intmaj;  /* internal maj devnum from master file*/
```

Figure E–4 doc_ Global Data Structure Declarations (*page 6 of 6*)

doc_init Driver Entry Point Routine

The initialization entry point routine performs the following tasks

- Sets up virtual-to-physical address translation for each configured controller (lines 301 – 312).
- Finds the external major number for each controller (lines 316 – 318) and determines the default parameters for each subdevice (lines 324 – 378). These parameters are initialized for each subdevice in lines 422 – 427 with a call to the subordinate driver routine, **doc_initdr**. Note the use of case statements (defined in the table in the master file) to handle different subdevice types (HARD, FLOPPY, STREAM, or NODRIVE) on the controller.
- Resets each controller and sets its interrupt vector to match that in the system's interrupt vector table generated by **lboot** (lines 383 – 415).
- Sets track buffer addresses (lines 429 – 457) and enable auto-flushing of those buffers (lines 460 – 481).
- Verifies status of controllers. Check for correct number of subdevices (lines 488 – 499) and if initialization of each is complete (lines 502 – 539). The polling for completion is necessary because an initialization routine cannot use the **sleep/wakeup** pair to synchronize hardware and software events. An alternate method for doing this check is to use the **delay** function.

Note that the header file defines the variables used for accessing the device, such as DOC_GOFLAG and DOC_COMMAND.

```
283  /*-------------------------------------------------------------*/
284  /*    initialization routine called once
285   *    during system startup,
286   */
287  doc_init()
288  {
289      register struct doc_00 *addr;
290      register int con, unit, subd, pi, j;
291      int vector, extmaj;
292      extern int hdeeduc, hdeedct;
293      dev_t ddev;
294      struct pdsector *pd;
295      DTRACE(" doc_init: start; tk buf %s\n",
                  (TBUFFER ? "ON" : "OFF"));
296  /*
297   *    set up each controller's address translation from kernel
298   *    virtual to VME physical, using sptalloc.  Virtual
299   *    addresses are in doc_caddr; physical in doc_cpaddr.
300   */
301      for (con=0; con < doc_numcontr; con++) {
302          doc_caddr[con] = sptalloc(btoc(2048),(PG_P|PG_LOCK),
303                                    btoc(doc_cpaddr[con]),0);
304          DEBUGinit (" doc_init: controller %d doc_caddr[]==0x%x\n",
305              con, doc_caddr[con]);
306          if (doc_caddr[con] == NULL) {
307              cmn_err(CE_WARN,
308             "doc_: sptalloc on controller %d failed.
                          Do not use device.\n",
309                        con);
310              return;
311          }
312      } /* for all controllers */

313  /*
314   *    find the controller's external major number
315   */
316      for (j=0; j<128; j++)
317          if (MAJOR[j] == doc_intmaj) break;
318      extmaj = j;
```

Figure E–5 doc_init Entry Point Routine (*part 1 of 8*)

```
319   /*
320    *     set up each unit's pointer block and initialize the device
321    *         with default parameters; these parameters will be changed
322    *         when the physical descriptor is read in on first open
323    */

324         for (unit=0; unit < doc_numcontr*4; unit++) {
325             con = contr(unit);
326             doc_tbufon[unit] = TBUFFER ;  /* tbuf is on for this unit */
327             doc_info[unit] = INFO_NULL;
328             doc_retrys[unit] = 0;
329             doc_tab[con].b_dev = makedev(extmaj,(unit<<4));
330             doc_tab[con].io_stp = &doc_iostat[unit];
331             switch (doc_itype[unit%4]) {
332             case DT_NODRIVE:
333                 doc_type[unit] = DT_NODRIVE;
334                 continue;
335             case DT_HARD:
336                 doc_type[unit] = DT_HARD;
337                 if (!doc_hard(unit)){
338                     cmn_err(CE_WARN,
339                       "doc_: controller %d drive %d cannot be
                                initialized as hard disk--ignored.\n",
340                         con,subdev(unit));
341                     doc_type[unit] = DT_NODRIVE;
342                     continue;
343                 }
344                 break;
345             case DT_FLOPPY:
346                 doc_type[unit] = DT_FLOPPY;
347                 if (doc_hard(unit)) {
348                     cmn_err(CE_WARN,
349                        "doc_: controller %d drive %d cannot
                                be initialized as floppy disk--ignored.\n",
350                         con,subdev(unit));
351                     doc_type[unit] = DT_NODRIVE;
352                     continue;
```

Figure E–5 doc_init Entry Point Routine (*part 2 of 8*)

```
353                 }
354                 break;
355             case DT_STREAMING:
356                 doc_type[unit] = DT_STREAMING;
357                 if (unit%4 != 1) {
358                     cmn_err(CE_WARN,
359                        "doc_: controller %d drive %d
                       cannot be initialized as stream tape--ignored.\n",
360                         con,subdev(unit));
361                     doc_type[unit] = DT_NODRIVE;
362                     continue;
363                 }
364                 break;
365             default:
366                 doc_type[unit] = DT_NODRIVE;
367                 continue;
368             }
369             pd = &doc_pdsect[unit];
370             if (doc_type[unit] == DT_HARD) {
371                     /* just enough to be able to read the real PDsect */
372                     pd->pdinfo.cyls = 1;
373                     pd->pdinfo.tracks = 1;
374                     pd->pdinfo.sectors = 18;
375                     pd->pdinfo.bytes = 512;
376             } else
377                 pd->pdinfo = floppy_pdsect;

378         } /* end for all units (all controllers) */
379     /*  for each controller, reset it and then set its
380      *  interrupt vector.  lboot initializes interrupt
381      *  vectors to be 16 * the external major number
382      */

383         for (con=0 ; con<doc_numcontr; con++) {
384             /* reset controller */
385             DEBUGinit(" doc_init: resetting %d\n",con);
```

Figure E–5 doc_init Entry Point Routine (*part 3 of 8*)

```
386                     if (doc_gocheck(con)) {
387                                 cmn_err(CE_WARN,
388                                         "doc_init: controller
                                    error: go-flag not clear\n");
389                                 cmn_err(CE_WARN,
390                                         "doc_init: before initial
                                    reset--don't use doc_\n");
391                                 return;
392                     }
393             DOC_COMMAND(con) = CMD_RESET;
394             DOC_GOFLAG(con) = GO_START;
395             if(doc_gocheck(con)) {
396                                 cmn_err(CE_WARN,
397                                         "doc_init: go not clear
                                    after reset don't use doc_\n");
398                                 return;
399                     }
400             if(DOC_ERRCODE(con) != ERR_NOERROR) {
401                                 cmn_err(CE_WARN,
402                                         "doc_init: 'reset
                                   controller' failed errcode==0x%x\n",
403                     DOC_ERRCODE(con));
404                                 cmn_err(CE_WARN,"doc_init: don't
                                      use doc_\n");
405                                 return;
406                     }

407     /* set controller interrupt vector */
408             for (j=0; j<128; j++)
409                 if (MAJOR[j] == doc_intmaj && MINOR[j] == 4*con) {
410                     vector = j << 4;
411                     break;
412             }
413             DOC_IVECTOR(con) = vector;
414             doc_fmtflag[con] = FMT_IDLE;

415         } /* for all controllers */
```

Figure E–5 doc_init Entry Point Routine (*part 4 of 8*)

```
416   /*
417    *    for each controller, initialize
418    *     drive parameters to those set above, set track
419    *     buffer addresses (4 per controller) and enable
420    *     auto-flushing of track buffers (once per controller).
421    */
422       for (con=0 ; con<doc_numcontr; con++) {
423           for (subd=0; subd<4; subd++) {

424    /* do "initialize drive" command, polling for completion */
425            unit = (con*4) + subd;
426            if (doc_initdr(unit))
427                    return;
428           } /* end for all subdv */
429           for (subd=0; subd<NTB; subd++) {

430    /* do "initialize track buffer" cmd, polling for completion */
431                /* error if go-flag says controller is busy */
432                if (doc_gocheck(con)) {
433                                              cmn_err(CE_WARN,
434                     "doc_init: controller error: go-flag not clear\n");
435                                              cmn_err(CE_WARN,
436          "doc_init: before init trk buf--don't use doc_\n");
437                                              return;
438                                    }
439          /* set command */
440         /* first two track buffer addresses are for hard disks*/
441                DOC_COMMAND(con) = ((subd<2) ?
                              CMD_HDIO : CMD_FLIO) | CMD_INITTB;
442                DOC_TBADDR_H(con) = hihalf(tbaddr[subd]);
443                DOC_TBADDR_L(con) = lohalf(tbaddr[subd]);
444                DEBUGinit(" doc_init: 'init track buffer'\n");
445                DOC_GOFLAG(con) = GO_START;
446                                   if(doc_gocheck(con)) {
447                                         cmn_err(CE_WARN,
448                                         "doc_init: go not clear after
                                     init trkbuf don't use doc_\n");
449                                         return;
450                                   }
```

Figure E–5 doc_init Entry Point Routine (*part 5 of 8*)

```
451                         if(DOC_ERRCODE(con) != ERR_NOERROR) {
452                            cmn_err(CE_WARN,
453                            "doc_init: init trkbuf failed
                                   errcode==0x%x don't use doc_\n",
454                             DOC_ERRCODE(con));
455                            return;
456                                }
457          } /* end for all subdv */

458   /* enable auto-flushing for hard disks on this  */
459   /* controller. Error if go-flag says controller is busy */

460                   if (doc_gocheck(con)) {
461                               cmn_err(CE_WARN,
462                               "doc_init: controller error:
                               go-flag not clear\n");
463                               cmn_err(CE_WARN,
464                               "doc_init: before enable
                               autoflush--don't use doc_\n");
465                               return;
466                   }
467          DOC_COMMAND(con) = CMD_ENBAUTOFL;
468          DEBUGinit(" doc_init: 'enable auto-flush'\n");
469          DOC_GOFLAG(con) = GO_START;
470                   if(doc_gocheck(con)) {
471                               cmn_err(CE_WARN,
472                               "doc_init: go not clear after
                               enab autoflush don't use doc_\n");
473                               return;
474                   }
475                   if(DOC_ERRCODE(con) != ERR_NOERROR) {
476                               cmn_err(CE_WARN,
477                               doc_init: enab autoflush failed
                              errcode==0x%x don't use doc_\n",
478                                  DOC_ERRCODE(con));
479                               return;
480                   }
481       } /* end for all controllers */
```

Figure E–5 doc_init Entry Point Routine (*part 6 of 8*)

```
482   /*  To verify that the controller is equipped with
483    *  the correct number of drives, do a "get status"
484    *  and check the results.  Use the true number of
485    *  sectors per track to determine block offsets
486    *  of partitions for floppies
487    */

488       for (unit=0 ; unit<doc_numcontr*4 ; unit++) {
489           switch (doc_type[unit]) {
490           case DT_NODRIVE:
491               break;
492           case DT_STREAMING:
493           case DT_FLOPPY:
494               pd = &doc_pdsect[unit];
495               for (j=0; j<IF_NUMPAR; j++)
496                   doc_vtoc[unit].v_part[j] = floppy_sizes[j];
497               doc_info[unit] = INFO_EQUIPPED;
498               break;

499           case DT_HARD:

500   /* do "get status" command, polling for completion */
501   /* error if go-flag says controller is busy */

502               if (doc_gocheck(contr(unit))) {
503                                           cmn_err(CE_WARN,
504                   "doc_init: controller error:
                              go-flag not clear\n");
505                                           cmn_err(CE_WARN,
506                   "doc_init: before get status
                              --don't use doc_\n");
507                                           return;
508                   }
```

Figure E–5 doc_init Entry Point Routine (*part 7 of 8*)

```
509                 DOC_COMMAND(contr(unit)) = CMD_HDIO | CMD_STATUS;
510     #ifdef DRIVETMP
511                 if (subdev(unit)==3) DOC_DRIVENO(contr(unit)) = 1 ;
512                         else  DOC_DRIVENO(contr(unit)) = subdev(unit);
513     #else
514                 DOC_DRIVENO(contr(unit)) = subdev(unit);
515     #endif
516                 DEBUGinit(" doc_init: 'get status' on %d\n",unit);
517                 DOC_GOFLAG(contr(unit)) = GO_START;
518                                 if(doc_gocheck(contr(unit))) {
519                                  cmn_err(CE_WARN,
520                                    "doc_init: go not clear after
                                          get status don't use doc_\n");
521                                   return;
522                                 }
523                 if (DOC_ERRCODE(contr(unit)) == ERR_NOERROR) {
524                     doc_info[unit] = INFO_EQUIPPED;
525                     DPRINT(" doc_init: unit %d equipped\n", unit);
526                 }
527                 else DPRINT(" doc_init: unit %d not equipped\n", unit);
528             } /* end switch */
529         } /* end for all units (all controllers) */
530     /*
531      *    Initialize bad block driver for each equipped drive
532      */
533         for (unit=0; unit<4*doc_numcontr; unit++)
534         if (doc_type[unit]==DT_HARD && doc_info[unit]&INFO_EQUIPPED) {
535                 ddev = makedev(extmaj, idmkmin(unit));
536                 hdeeqd(ddev, IDPDBLKNO, EQD_ID);
537         }
538         DTRACE(" doc_init: return\n");

539     } /* end init */
```

Figure E–5 doc_init Entry Point Routine *(part 8 of 8)*

doc_initdr Subordinate Driver Routine

This subordinate driver routine is called by the **doc_init** entry point routine to actually initialize the subdevices of the controllers. You may have noticed the comment (lines 26 – 29) that explains why this is now in a subordinate routine. Because this is a part of the driver that interacts directly with the device itself, it makes good sense to isolate it in a subroutine; should this code be rewritten at a later date to support another device (or an enhanced version of this device), this subordinate routine may need to be rewritten but other parts of the initialization routine will not.

Note how this routine utilizes the variables that are defined in the header file (lines 571 – 576; see the header file, lines 40 – 45) for accessing the subdevices.

```
540   /*-------------------------------------------------------*/
541   /*
542    *  doc_initdr - Initialize drive parameters in controller.
543    *                Used whenever pdsect is changed.
544    *                Return 1 if failure, 0 if success.
545    */
546   static int
547   doc_initdr(unit)
548        int unit;
549   {
550        int con, subd;

551        con = contr(unit);
552        subd = subdev(unit);

553        /* error if go-flag says controller is busy */
554        if (doc_gocheck(con)) {
555                  cmn_err(CE_WARN,
556                  "doc_initdr: controller error:
                                 go-flag not clear\n");
557                  cmn_err(CE_WARN,
558                  "doc_initdr: before init
                                 drive--don't use doc_\n");
559                  return(1);
560        }

561        DOC_COMMAND(con) = CMD_INITDR
562        | ((doc_type[subd] == DT_HARD) ? CMD_HDIO : CMD_FLIO);
```

Figure E–6 doc_initdr Subordinate Driver Routine (*part 1 of 2*)

```
563   #ifdef DRIVETMP
564        if (subd==3)
565                   DOC_DRIVENO(con) = 1 ;
566        else
567                   DOC_DRIVENO(con) = subd;
568   #else
569        DOC_DRIVENO(con) = subd;
570   #endif

571   DOC_NHEADS(con)  = (u_char)(doc_pdsect[(4*con)+subd].pdinfo.tracks);
572   DOC_MAXCYL(con)  = (u_short)(doc_pdsect[(4*con)+subd].pdinfo.cyls-1);
573   DOC_NSECTRK(con) = (u_char)(doc_pdsect[(4*con)+subd].pdinfo.sectors);
574   DOC_NBYTSEC(con) = (u_short)(doc_pdsect[(4*con)+subd].pdinfo.bytes);
575        if (doc_type[subd] == DT_HARD)
576                   DOC_HDGAP(con) = HDG_512;

577        DEBUGinit(" doc_initdr: 'init drive' on %d\n", con);
578        DOC_GOFLAG(con) = GO_START;

579        if(doc_gocheck(con)) {
580                   cmn_err(CE_WARN,
581                   "doc_initdr: go not clear after
                        init drive don't use doc_\n");
582                   return(1);
583        }

584        if(DOC_ERRCODE(con) != ERR_NOERROR) {
585                   cmn_err(CE_WARN,
586                                "doc_initdr: init drive failed
                              errcode==0x%x don't use doc_\n",
587                                DOC_ERRCODE(con));
588                   return(1);
589        }
590        return(0);
591   }
```

Figure E–6 doc_initdr Subordinate Driver Routine (*part 2 of 2*)

doc_open Driver Entry Point Routine

The **doc_** driver does some further initialization of the device the first time it is opened. This enables it to use the file system to download physical description, vtoc, and defect information to the disk.

Before doing any initialization, the **open** routine checks that the device is there (lines 610 – 614), that no other **opens** are executing against the device (lines 618 – 620), that this is the first **open** of the device since boot (lines 624 – 626), and that the unit is equipped with a hard disk (lines 638 – 644).

Note how the physical descriptor sector is read into a buffer (lines 648 – 657) using the **doc_strategy** routine (line 651), **iowait** (line 652) to acquire the information, and the subordinate static routine **doc_copy** (line 657) to move it into a local variable on the stack. A similar approach is used to read in the defect map (lines 676 – 691) and the VTOC (lines 695 – 707).

```
592   /*-----------------------------------------------------*/
593   /*
594    *  doc_open - on first open read in physical
595    *             description, vtoc, and defect info
596    */
597   /*ARGSUSED*/
598   doc_open(dev,flag,otyp)
599   {
600       struct buf *geteblk();
601       struct buf *bufhead;
602       register int unit, defcnt;
603       int defaddr;
604       struct pdsector *pd;

605       DTRACE(" doc_open: dev %d flag %d otyp %d\n", dev,flag,otyp);
606       unit = iddn(minor(dev));
607   /*
608    *    Make sure there is a device there
609    */
610       if (!(doc_info[unit]&INFO_EQUIPPED)) {
611           /* no disk out there */
612           u.u_error = ENXIO;
613           return;
614       }
```

Figure E–7 doc_open Routine *(part 1 of 6)*

```
615   /*
616    *    Wait for any other open to complete
617    */

618       while (doc_info[unit]&INFO_OPENING) {
619           sleep(&doc_info[unit],PZERO);
620       }

621   /*
622    *    For the first open do all the hard work
623    */

624       if (!(doc_info[unit]&INFO_OPEN)) {
625           doc_info[unit] |= INFO_OPENING;

626           pd = &doc_pdsect[unit];
627   /*
628    *    initialize defect tables
629    */

630           for(defcnt=0; defcnt<(DEFCNT); defcnt++) {
631               doc_defect[unit].map[defcnt].bad.full = 0xffffffff;
632               doc_defect[unit].map[defcnt].good.full = 0xffffffff;
633           }

634   /*
635    *    if the unit is not equipped with a hard disk, skip reading the
636    *        pdsect, vtoc and bad block info
637    */

638           if (doc_type[unit] != DT_HARD) {
639               doc_info[unit] |= INFO_OPEN;
640               doc_vtoc[unit].v_sanity |= VTOC_SANE;
641               doc_info[unit] &=  INFO_OPENING;
642               wakeup(&doc_info[unit]);
643               return;
644           }
```

Figure E–7 doc_open Routine (*part 2 of 6*)

```
645   /*
646    *     read physical description sector
647    */

648           bufhead = geteblk();
649           doc_setblk (bufhead, B_READ, IDPDBLKNO, dev);
650           bufhead->b_bcount = pd->pdinfo.bytes;
651           doc_strategy(bufhead);
652           iowait(bufhead);
653           if (bufhead->b_flags&B_ERROR)     {
654               cmn_err(CE_WARN,
                        "doc_: Cannot read physical descriptor
                        sector on controller %d,
                        drive %d.\n",contr(unit),subdev(unit));
655               goto badopen;
656           }
657           doc_copy (bufhead->b_un.b_addr, pd, sizeof(struct pdsector));

658   /*
659    *     If it wasn't valid undo the damage
660    */

661            if (pd->pdinfo.sanity != VALID_PD) {
662               cmn_err(CE_WARN, "doc_:  Bad physical
                        descriptor sanity word on controller %d,
                                drive %d.\n",contr(unit),subdev(unit));
663         /* just enough to be able to read the real PDsect */
664                    pd->pdinfo.cyls = 1;
665                    pd->pdinfo.tracks = 1;
666                    pd->pdinfo.sectors = 18;
667                    pd->pdinfo.bytes = 512;
668                doc_initdr(unit);  /* re-initialize controller */
669               goto badopen;
670           }

671         if (doc_initdr(unit))   /* re-initialize controller */
672               goto badopen;
```

Figure E–7 doc_open Routine (*part 3 of 6*)

```
673   /*
674    *     read the defect map
675    */
676             if (pd->pdinfo.defectsz > DEFSIZ) {
677                 cmn_err (CE_WARN,
                            "doc_: Too little space allocated
                            in driver for defect table on controller %d,
                            drive %d\n", contr(unit),subdev(unit));
678                 goto badopen;
679             }
680             for (defcnt=0; defcnt <
                     (pd->pdinfo.defectsz/pd->pdinfo.bytes); defcnt++) {
681                 doc_setblk (bufhead, B_READ,
                        pd->pdinfo.defectst+defcnt, dev);
682                 bufhead->b_bcount = pd->pdinfo.bytes;
683                 doc_strategy(bufhead);
684                 iowait(bufhead);
685                 if (bufhead->b_flags & B_ERROR)     {
686                     cmn_err(CE_WARN, "doc_: Cannot read defect
                           map on controller %d, drive %d\n",
                           contr(unit),subdev(unit));
687                     goto badopen;
688                 }
689                 defaddr = ((int)&doc_defect[unit]) +
                                (defcnt*pd->pdinfo.bytes);
690                 doc_copy (bufhead->b_un.b_addr, defaddr,
                                pd->pdinfo.bytes);
691             }
```

Figure E–7 doc_open Routine (*part 4 of 6*)

```
692   /*
693    *     read in the vtoc
694    */
695             doc_setblk (bufhead,B_READ,
                          pd->pdinfo.logicalst+IDVTOCBLK,dev);
696             bufhead->b_bcount = pd->pdinfo.bytes;
697             doc_strategy(bufhead);
698             iowait(bufhead);
699             if (bufhead->b_flags & B_ERROR)     {
700                 cmn_err(CE_WARN, "doc_:  Cannot read VTOC
                        on controller %d, drive %d\n",contr(unit),
                        subdev(unit));
701                 goto opendone;
702             }
703             doc_copy (bufhead->b_un.b_addr,
                      &doc_vtoc[unit],sizeof(struct vtoc));
704             if (doc_vtoc[unit].v_sanity != VTOC_SANE)     {

705                 cmn_err(CE_WARN, "doc_: Bad sanity word in
                      VTOC on controller %d, drive %d.\n",
                       contr(unit),subdev(unit));
706                 goto opendone;
707             }
```

Figure E–7 doc_open Routine (*part 5 of 6*)

```
708   /*
709    *    open is complete - wakeup sleeping processes and return buffer
710    */

711           doc_info[unit] |= INFO_OPEN;
712           goto opendone;

713   /*  If the open was for a physical device (whole drive) but
714    *  the open was bad, mark the drive as open anyway.  This
715    *  is so the drive can be opened even though no
716    *  information has been written to the disk; thus an
717    *  ioctl call can be used to format the disk.
718    */

719   badopen:
720           if (!idnodev(minor(dev))){
721               u.u_error = ENXIO;
722           }
723           else {
724               doc_info[unit] |= INFO_OPEN;
725               u.u_error = 0;
726           }

727   opendone:
728           doc_info[unit] &=  INFO_OPENING;
729           wakeup(&doc_info[unit]);
730           bufhead->b_flags |= B_ERROR;     /* mark the buffer bad */
731           brelse(bufhead);
732       }
733   DTRACE(" doc_open: return\n");
734   } /* end doc_open */
```

Figure E–7 doc_open Routine (*part 6 of 6*)

doc_close Driver Entry Point Routine

The **doc_close** entry point routine is an empty routine. An installed driver must have an entry in the switch table for the close routine, but this device requires no special action.

Lines 746 – 748 restore the names of three buffer-header members to ensure that they are accessible by another process. Table E-3 summarizes these members and where they are used in the driver code.

Table E–3 Buffer Header Members Restored by doc_close Routine

Member	*Header File*	*Where used in doc_ (line numbers)*
b_resid	*sys/buf.h*	*as* **cylin**, 832, 835, 868, 870
io_s1	*sys/iobuf.h*	as **acts**, 855, 867, 1310, 1311
jrqsleep	*sys/iobuf.h*	a counter that is modified indirectly

```
735   /*-------------------------------------------------------*/
736   /*
737    *    doc_close - provided as standard interface
738    */

739   doc_close()
740   {
741   }

742   /*
743    *    Change the names of things in buffers
744    *    and buffer headers for different uses
745    */

746   #define cylin   b_resid
747   #define acts    io_s1
748   #define ccyl    jrqsleep
```

Figure E–8 doc_close Entry Point Routine

doc_strategy Driver Entry Point Routine

The **doc_strategy** entry point routine is responsible for the actual I/O transfer when doing block-access for the device. Note that this same routine is accessed as a subordinate routine when doing character-access of the device (see line 1340) and when reading the physical description sector, defect map, and device vtoc in the **doc_open** routine (lines 651, 683, and 697).

The **doc_strategy** routine does a series of checks (lines 765 – 824), collects some information needed to do and track the transfer (lines 826 – 843), puts the buffer header in the queue (lines 847 – 879), and calls the subordinate routine, **doc_iostart** (line 856) to do the actual I/O operation. The **diskerr** subroutine (lines 888 – 892) is called if any of the checks in the **doc_strategy** routine fail.

```
749   /*-------------------------------------------------------*/
750   /*    Device strategy routine: do partition
751    *      checks, sort I/O queue, and so on
752    */
753   doc_strategy (bufhead)
754   register struct buf *bufhead;
755   {
756       register struct iobuf *drvtab; /* drive status pointer */
757       register struct pdsector *pd;  /* pointer to phys desc */
758       register int unit;      /* drive unit ID */
759       daddr_t lastblk;        /* last block in partition */
760       int partition;          /* drive partition number */
761       int iplsave;            /* saved interrupt level */
762       int sectoff;            /* start sector of partition */
763       int mdev;               /* minor dev num of device */
764   /*
765    *    Decode the device number
766    */
767       mdev = minor(bufhead->b_dev);
768       partition = idslice(mdev);
769       unit = iddn(mdev);
770       DTRACE(" doc_strategy: mdev %d partition %d
              unit %d\n",mdev,partition,unit);
```

Figure E–9 doc_strategy Driver Entry Point Routine (*part 1 of 5*)

```
771   /*
772    *    Check to see if there is really a device there
773    */
774       if (!(doc_info[unit]&INFO_EQUIPPED)) {
775           goto diskerr;
776       }

777   /*    Get the device physical information and pick
778    *    up the partition beginning and end.
779    *    The whole disk (idnodev) is a special case.
780    */
781       pd = &doc_pdsect[unit];
782       if (idnodev(mdev)) { /* writing on whole disk */
783           lastblk = (pd->pdinfo.sectors * pd->pdinfo.tracks *
784                  pd->pdinfo.cyls);
785           sectoff = 0x00;
786       } else {
787   /*
788    *    check for invalid VTOC
789    */
790           if (doc_vtoc[unit].v_sanity != VTOC_SANE) {
791               goto diskerr;
792           }
793   /*
794    *    check for read only partition
795    */
796           if ((
                    (doc_vtoc[unit].v_part[partition].p_flag&V_RONLY)
                       == V_RONLY)
797                && ((bufhead->b_flags&B_READ) != B_READ)) {
798               u.u_error = ENXIO;
799               cmn_err (CE_WARN, "doc_: partition %d on
                        controller %d, drive %d is marked read only\n",
                          partition, contr(unit),subdev(unit));
800               goto diskerr;
801           }
802           lastblk = doc_vtoc[unit].v_part[partition].p_size;
803            sectoff = (doc_vtoc[unit].v_part[partition].p_start
804                + pd->pdinfo.logicalst);
805       }
```

Figure E–9 doc_strategy Driver Entry Point Routine (*part 2 of 5*)

```
806   /*
807    *    Get the queue header
808    */
809       drvtab = &doc_tab[contr(unit)];
810   /*    Check to see if the requested block exists
811    *    within requested partition
812    */
813       if ((bufhead->b_blkno +
                ((bufhead->b_bcount-1)/pd->pdinfo.bytes)>=lastblk)
814           || (bufhead->b_blkno < DOC_FRSTBLK)) {
815           if ((bufhead->b_blkno==lastblk) &&
                    (bufhead->b_flags&B_READ)) {
816   /*
817    *    Make eof on read work correctly
818    */
819               bufhead->b_resid = bufhead->b_bcount;
820               iodone(bufhead);
821               return;
822           }
823           goto diskerr;
824       }
825   /*    ENTER CRITICAL REGION - spl5 = 10 on
826    *    the processor = spl5 on the VMEbus
827    */
828       iplsave = spl5();
```

Figure E–9 doc_strategy Driver Entry Point Routine (*part 3 of 5*)

```
829   /*
830    *    store the cylinder number for disk sort
831    */
832        bufhead->cylin = ((bufhead->b_blkno+sectoff) /
833           (pd->pdinfo.sectors*pd->pdinfo.tracks));
834       DEBUGnums(" doc_strategy: bufhead->b_blkno,
                    bufhead->cylin==%d,%d\n",
835          bufhead->b_blkno,bufhead->cylin);
836   /*
837    *    Collect some statistics
838    */
839       bufhead->b_start = lbolt; /* time stamp request */
840       doc_time[unit].io_cnt++;    /* inc operations count */
841       doc_time[unit].io_bcnt +=
                (bufhead->b_bcount+pd->pdinfo.bytes-1)
842                    pd->pdinfo.bytes;
843       drvtab->qcnt++;   /* inc drive current request count */

844   /*
845    *    Put the buffer header in the queue
846    */
847       bufhead->av_forw = DOC_NULL;
848       if (drvtab->b_actf == DOC_NULL) {
849   /*
850    *    If the queue is empty, just put it at the
851    *        head and then call the start IO routine
852    */
853          drvtab->b_actf = bufhead;
854          drvtab->b_actl = bufhead;
855          drvtab->acts = (int)bufhead;
856          doc_iostart(unit);
857       } else
```

Figure E–9 doc_strategy Driver Entry Point Routine (*part 4 of 5*)

```
858   /*
859    *    Otherwise we do a disk sort to figure
860    *     out where to put the buffer on the queue
861    */
862         {
863            register struct buf *ap, *cp;
864            int s1, s2;

865            if (((int)doc_time[unit].io_cnt&0x0f) == 0)
866                drvtab->acts = (int)drvtab->b_actl;
867            for (ap=(struct buf *)drvtab->acts; cp=ap->av_forw; ap=cp) {
868                if ((s1 = ap->cylin - bufhead->cylin)<0)
869                    s1 = -s1;
870                if ((s2 = ap->cylin - cp->cylin)<0)
871                    s2 = -s2;
872                if (s1 < s2)
873                    break;
874            }
875            ap->av_forw =  bufhead;
876            if ((bufhead->av_forw = cp) == DOC_NULL)
877                drvtab->b_actl = bufhead;
878            bufhead->av_back = ap;
879       }
880   /*
881    *   .FGIT CRITICAL REGION
882    */
883       splx (iplsave);
884       return;
885   /*    If an error occurs wake up who ever is
886    *      waiting so they can get an error
887    */
888   diskerr:
889       bufhead->b_flags |= B_ERROR;
890       bufhead->b_error = ENXIO;
891       iodone (bufhead);
892       return;
893   } /* end strategy */
```

Figure E–9 doc_strategy Driver Entry Point Routine (*part 5 of 5*)

doc_iostart Subordinate Driver Routine

The **doc_iostart** routine provides the device-specific interaction necessary for the I/O transfer. It is called by the **doc_strategy** routine to start the I/O transfer and by the **doc_int** routine to handle the job completion interrupt generated when the I/O transfer is completed. The controller associated with this driver has the intelligence to handle much of the I/O transfer itself; isolating the code that intimately interacts with the intelligent firmware is a good programming practice that enhances both the portability and maintainability of the driver.

Note the use of variables for interfacing with the hardware that are defined in the driver's header file. Should a new version of the hardware require modification of these values, they can be redefined in the header file without recoding the driver.

```
894   /*-------------------------------------------------------*/
895   /*  start a disk I/O, this must called with disk
896    *  interrupts disabled.  Set up parameters for
897    *  controller and start command.  It is called
898    *  from two places, the strategy routine when a
899    *  buffer is put onto an empty queue, and after
900    *  an I/O completes in the interrupt routine.
901    */

902   static
903   doc_iostart(unit)
904   register int unit;
905   {
906       register struct buf *bp;    /* pointer to buffer header */
907       register struct iobuf *dp; /* pointer to queue header */
908       register int i;             /* temporary */
909       register struct defect *deftab; /* pointer of defect table */
910       register struct pdsector *pd; /* pointer to physical info */
911       int  firstbn;               /* block number of job start */
912       int  cylsize;               /* temp, num of blks in a cyl */
913       long paddress;              /* buffer address */
914       long addr;                  /* buffer address */
915       union diskaddr firstsect;  /* the first sector in the IO */
```

Figure E–10 doc_iostart Subordinate Routine (*part 1 of 5*)

```
916   /*
917    *    Get the queue header
918    */
919       dp = &doc_tab[contr(unit)];
920   /*
921    *    Pull the buffer from the start of the list.
922    *        If there is no work to do, or if a format
923    *        is in progress, just return.
924    *
925    *    Note: a format on any one unit of a controller
926    *        occupies that controller totally.  Jobs
927    *        for any other unit on that controller just
928    *        pile up in the queue until the format finishes.
929    */
930       if (doc_fmtflag[contr(unit)] != FMT_IDLE) {
931           return;
932       }
933       bp=dp->b_actf;
934       if (bp == DOC_NULL) {
935           wakeup(dp);     /* wake up any formatting request */
936           return;
937       }
938   /*
939    *  all the requests for any unit on the same controller
940    *  are in the same queue. When we get new entries from the
941    *  queue we have to recompute the unit number ...
942    */
943       unit = iddn(minor(bp->b_dev));
944   /*
945    *    set up pointers to relevant data structures.
946    *        Now we have a context for the IO
947    */
948       deftab = doc_defect[unit].map;
949        pd = &doc_pdsect[unit];
950   /*
951    *    calculate the true block number from the partition offset
952    */
953       firstbn = bp->b_blkno;
954       if (!idnodev(minor(bp->b_dev))) {
955           firstbn +=
              doc_vtoc[unit].v_part[idslice(minor(bp->b_dev))].p_start
956               + pd->pdinfo.logicalst;
957       }
```

Figure E–10 doc_iostart Subordinate Routine *(part 2 of 5)*

```
958         DEBUGnums(" doc_iostart: bp->b_blkno==%d; real firstbn==%d\n",
959             bp->b_blkno,firstbn);
960     /*
961      *    get physical address from buffer header
962      */
963         paddress = vtop((int)bp->b_un.b_addr, bp->b_proc);
964         if (paddress == DOC_NULL) {
965             cmn_err(CE_PANIC,"doc_: Bad address returned by VTOP\n");
966             return;
967         }
968         cylsize = pd->pdinfo.tracks * pd->pdinfo.sectors;
969     /*
970      *    on the first time around set the residual correct
971      *        and time stamp it
972      */
973         if (dp->b_active == 0) {
974             bp->b_resid = bp->b_bcount;
975             doc_count[unit] = 0;
976             dp->b_active++;
977             dp->io_start = lbolt;
978         }
979     /*
980      *    don't transfer more than (pd->pdinfo.bytes) bytes at
981      *     once because this is a one-block-at-a-time controller.
982      */
983         doc_tcount[unit] = (bp->b_resid > pd->pdinfo.bytes
984                     ? pd->pdinfo.bytes : bp->b_resid);
985     /*    compute disk address
986      *      1) get the first block of this IO
987      *      2) convert it to the units of the device (128/256/512)
988      *      3) figure out block after the last one in the job
989      *      4) calculate the values for the sector/head/tracks
990      *
991      *    first block number in terms of this
992      *      device's physical sectors              */
993         firstbn += (doc_count[unit] >> 9);
```

Figure E–10 doc_iostart Subordinate Routine (*part 3 of 5*)

```
 994   /*
 995    *    look for bad blocks for this job
 996    *        (but only for hard disks )
 997    */
 998       if (doc_type[unit] != DT_HARD) {
 999           goto startcmd; /* no bad blocking for floppies! */
1000       }
1001   /* convert block number into disk-address format */

1002       firstsect.part.pcn = firstbn / cylsize;           /* cyl */
1003       i = firstbn % cylsize;
1004       firstsect.part.phn = i / pd->pdinfo.sectors;      /* head */
1005       firstsect.part.psn = i % pd->pdinfo.sectors;  /* sector */

1006   /* search defect map */
1007       for (i=0;
1008           ((i<DEFCNT) && (firstsect.full > deftab->bad.full))
1009                           ; i++)
1010           deftab++;
1011   /*   if there are any, then all that has to be done
1012    *   is to substitute the good block number for the
1013    *   bad one.  Since we only transfer one sector at
1014    *   a time, we don't have to worry about crossing
1015    *   over track boundaries and such.
1016    */
1017       if ((i<DEFCNT) && (firstsect.full == deftab->bad.full)) {
1018           DPRINT(" doc_iostart: defect hit; block %d
                       remapped\n",firstbn);
1019           firstbn = (deftab->good.part.pcn * cylsize)
1020                   + (deftab->good.part.phn * pd->pdinfo.sectors)
1021                   + (deftab->good.part.psn);
1022           DPRINT(" doc_iostart: defect remapped to
                         block %d\n",firstbn);
1023       }
```

Figure E–10 doc_iostart Subordinate Routine (*part 4 of 5*)

```
1024    /*
1025     *    set up the io packet and do it
1026     */
1027    startcmd:
1028        /* error if go-flag says controller is busy */
1029         if (doc_gocheck(contr(unit))) {
1030                     cmn_err(CE_WARN,
1031                      "doc_iostart: error: go-flag not clear
                          before iostart\n");
1032                     cmn_err(CE_WARN,
1033                    "doc_iostart: aborting i/o request\n");
1034                      return;
1035         }
1036        DOC_COMMAND(contr(unit)) =
1037            ((doc_type[unit]==DT_HARD) ? CMD_HDIO : CMD_FLIO)
1038            | (doc_tbufon[unit] ? 0 : CMD_FORCE)
                          /* force sing sec io after errs */
1039            | ((bp->b_flags&B_READ) ? CMD_READ : CMD_WRITE)
1040            | CMD_INTWD;          /* interrupt when done */
1041        doc_tbufon[unit] = TBUFFER ;
               /* always reset init tbuf condition */

1042        addr = VMEMEM(paddress+doc_count[unit]);
1043        DOC_SBADDR_H(contr(unit)) = hihalf(addr);
1044        DOC_SBADDR_L(contr(unit)) = lohalf(addr);

1045    startio:
1046    #ifdef DRIVETMP
1047         if (subdev(unit)==3) DOC_DRIVENO(contr(unit)) = 1 ;
1048         else       DOC_DRIVENO(contr(unit)) = subdev(unit);
1049    #else
1050        DOC_DRIVENO(contr(unit)) = subdev(unit);
1051    #endif
1052        DOC_LBN_H(contr(unit)) = hihalf(firstbn);
1053        DOC_LBN_L(contr(unit)) = lohalf(firstbn);
1054    /*
1055     *    poke the device to start the i/o; return immediately,
1056     *    so an interrupt coming soon after the go isn't lost
1057     */
1058        DOC_GOFLAG(contr(unit)) = GO_START;
1059    }
```

Figure E–10 doc_iostart Subordinate Routine (*part 5 of 5*)

doc_int Driver Interrupt Handler

The **doc_int** routine is the driver's interrupt handler. In this driver, it identifies which subdevice generated the interrupt (which is an operating system interface) then calls the **doc_intr** subordinate routine to service the actual interrupt. By separating the code that interracts with the device itself into a separate subroutine, the portability and maintainability of the driver code is enhanced.

```
1060   /*-------------------------------------------------------------*/
1061   /*
1062    *  the device interrupt service routine, figure out which
1063    *  disks have interrupted and call their service routines
1064    */
1065   doc_int(ivec)
1066   int ivec;
1067   {
1068   #ifdef DRIVETMP
1069       register int unit,drv;
1070   #else
1071        register int unit;
1072   #endif
1073   /*
1074    * ivec is the number of the controller that had the interrupt
1075    */
1076   #ifdef DRIVETMP
1077        if ( (drv=DOC_DRIVENO(ivec)) == 1 ) drv=3;
1078       unit = (4 * ivec) + drv;
1079   #else
1080       unit = (4 * ivec) + DOC_DRIVENO(ivec);
1081   #endif
1082       DPRINT(" doc_int: ivec 0x%x unit %d\n",ivec, unit);
1083       doc_intr(unit);
1084   }
```

Figure E–11 doc_int Driver Interrupt Handler

doc_intr Subordinate Driver Routine

The **doc_intr** routine handles any possible interrupt that could come from a subdevice.

```
1085   /*-------------------------------------------------------*/
1086   /*
1087    *  this routine is called from the one above when the
1088    *  unit(s) that caused the interrupt has been discovered
1089    */

1090   static
1091   doc_intr(unit)
1092   register int unit;
1093   {
1094       register struct buf *bp;
1095       register struct iobuf *dp;
1096       register int i;
1097       short prterr;
1098       u_char errcode;

1099       DTRACE(" doc_intr: start\n");
1100       dp = &doc_tab[contr(unit)];
1101       errcode = DOC_ERRCODE(contr(unit));
1102   /*
1103    *    handle formatting interrupt if format is in progress
1104    *    and was successful.
1105    */
1106       if ((doc_fmtflag[contr(unit)]==
                FMT_INPROGRESS) && (errcode==ERR_NOERROR))
1107       {
1108           DEBUGform(" doc_intr: format succeeded\n");
1109           doc_fmtflag[contr(unit)] =
                    FMT_SUCCEED; /* finished successfully */
1110           wakeup(&doc_fmtflag[contr(unit)]);
                        /* wake sleeping IOCTL*/
1111           return;
1112       }
1113       bp = dp->b_actf;
```

Figure E–12 doc_intr Subordinate Driver Routine (*part 1 of 9*)

```
1114    /*
1115     *     if not formatting, look for spurious interrupts
1116     */
1117        if (doc_fmtflag[contr(unit)] != FMT_INPROGRESS) {
1118            if (dp->b_active == 0)
1119                goto spurious;
1120            if (bp == 0) {
1121                dp->b_active = 0;
1122    spurious:
1123                cmn_err(CE_WARN, "doc_: Spurious interrupt
                for controller %d, drive %d\n",contr(unit),subdev(unit));
1124                return;
1125            }
1126        }
1127    /*
1128     *     now see if the previous io completed ok
1129     */
1130        if (errcode != ERR_NOERROR) {
1131            prterr = 0;
1132            switch (errcode) {
1133                case ERR_DNOTREADY:
1134                    cmn_err(CE_WARN,"doc_: controller %d,
                           drive %d Drive not ready\n",contr(unit),
                           subdev(unit));
1135                    break;
1136                case ERR_RESERVED:
1137                    cmn_err(CE_WARN, "doc_: controller %d,
                           drive %d Reserved error code returned\n",
                           contr(unit),subdev(unit));
1138                    break;
```

Figure E–12 doc_intr Subordinate Driver Routine (*part 2 of 9*)

```
1139                case ERR_ACCESSERR:
1140                    if (doc_fmtflag[contr(unit)] != FMT_INPROGRESS) {
1141                        doc_retrys[unit]++;
1142                        if (((doc_type[unit] == DT_HARD)
1143                            && (doc_retrys[unit] < HRETRYS))
1144                            || ((doc_type[unit] == DT_FLOPPY)
1145                            && (doc_retrys[unit] < FRETRYS)))
1146                            {
1147                              if (idnodev(bp->b_dev)) {
1148                                          /* access was "physical" */
1149                                  DEBUGretry(" doc_: controller %d,
                                   drive %d, phys block %d:
                                               retry - access error\n",
                                   contr(unit),subdev(unit),bp->b_blkno);
1150                                    } else {
1151                                          /* access was "logical" */
1152                                  i = bp->b_blkno
1153         + doc_vtoc[unit].v_part[idslice(minor(bp->b_dev))].p_start
1154         + doc_pdsect[unit].pdinfo.logicalst;
1155                                          DEBUGretry(" doc_: controller %d,
                        drive %d, partition %d, log block %d,
                            phys block %d: retry - access error\n",
                        contr(unit),subdev(unit),
                        idslice(minor(bp->b_dev)),bp->b_blkno,i);
1156                                    }
1157                              doc_tbufon[unit] = 0; /* turn off
                                              tbuf for retry */
1158                              doc_iostart(unit);
1159                              return;
1160                          }
1161                          doc_retrys[unit] = 0;
1162                          if (doc_type[unit]==DT_HARD) prterr++;
1163                      }
1164                     cmn_err(CE_WARN, "doc_: controller %d,
                            drive %d Disk access error\n",
                            contr(unit),subdev(unit));
1165                     break;
1166                case ERR_VERIFYERR:
1167                     cmn_err(CE_WARN, "doc_: controller %d,
                            drive %d Verify error\n",contr(unit),
                            subdev(unit));
1168                     break;
```

Figure E–12 doc_intr Subordinate Driver Routine *(part 3 of 9)*

```
1169                case ERR_DMAERR:
1170                    cmn_err(CE_WARN, "doc_: controller %d,
                            drive %d DMA error\n",contr(unit),
                            subdev(unit));
1171                    break;
1172                case ERR_DRVNOTINIT:
1173                    cmn_err(CE_WARN, "doc_: controller %d,
                            drive %d Drive or track buffer not
                            initialized\n",contr(unit),subdev(unit));
1174                    break;
1175                case ERR_NUMTBS:
1176                    cmn_err(CE_WARN, "doc_: controller %d,
                            drive %d Too many track buffers\n",
                            contr(unit),subdev(unit));
1177                    break;
1178                case ERR_ILLEGALCMD:
1179                    cmn_err(CE_WARN, "doc_: controller %d,
                            drive %d Illegal command\n",contr(unit),
                            subdev(unit));
1180                    break;
1181                case ERR_ILLEGALLBN:
1182                    cmn_err(CE_WARN, "doc_: controller %d,
                            drive %d Illegal block number\n",contr(unit),
                            subdev(unit));
1183                    break;
1184                case ERR_SEEKERR:  /* floppy only */
1185                    cmn_err(CE_WARN,"doc_: controller %d,
                            drive %d floppy seek error\n",contr(unit),
                            subdev(unit));
1186                    /* fall thru ! */
```

Figure E–12 doc_intr Subordinate Driver Routine (*part 4 of 9*)

```
1187                case ERR_CRCERR:  /* floppy only */
1188                    if (doc_fmtflag[contr(unit)] != FMT_INPROGRESS) {
1189                        doc_retrys[unit]++;
1190                        if (doc_retrys[unit] < FRETRYS) {
1191                              if (idnodev(bp->b_dev)) {
1192                                                  /* access was "physical" */
1193                                  DEBUGretry(" doc_: controller %d,
                                    drive %d, phys block %d:
                                                  retry - CRC error\n",contr(unit),
                                    subdev(unit),bp->b_blkno);
1194                                    } else {
1195                                                  /* access was "logical" */
1196                                    i = bp->b_blkno
1197         + doc_vtoc[unit].v_part[idslice(minor(bp->b_dev))].p_start
1198         + doc_pdsect[unit].pdinfo.logicalst;
1199                                                  DEBUGretry(" doc_: controller %d,
                      drive %d, partition %d, log block %d, phys block %d:
                      retry - CRC error\n",contr(unit),subdev(unit),
                        idslice(minor(bp->b_dev)),bp->b_blkno,i);
1200                                    }      /* turn off tbuf for retry */
1201                              doc_tbufon[unit] = 0;
1202                              doc_iostart(unit);
1203                              return;
1204                        }
1205                        doc_retrys[unit] = 0;
1206                    }
1207                    cmn_err(CE_WARN,"doc_: controller %d,
                        drive %d floppy CRC error\n",contr(unit),
                        subdev(unit));
1208                    break;

1209                case ERR_WRITEPROT:  /* floppy only */
1210                    cmn_err(CE_WARN,"doc_: controller %d,
                        drive %d Attempt to write on
                        write-protected media\n",contr(unit),
                        subdev(unit));
1211                    break;
```

Figure E–12 doc_intr Subordinate Driver Routine (*part 5 of 9*)

```
1212                case ERR_BADMEDIA:
1213                    cmn_err(CE_WARN,"doc_: controller %d,
                            drive %d Uninitialized or un-readable
                            media\n",contr(unit),subdev(unit));
1214                    break;
1215            }
1216    /*  If error occurred during formatting, just
1217     *   return error code to IOCTL and don't worry
1218     *    about error logging or specifics
1219     */
1220            if (doc_fmtflag[contr(unit)] == FMT_INPROGRESS) {
1221                DEBUGform(" doc_intr: format failed\n");
1222                doc_fmtflag[contr(unit)] =
                          FMT_FAIL;    /* finished and failed */
1223                wakeup(&doc_fmtflag[contr(unit)]);
                          /* wake sleeping IOCTL */
1224                return;
1225            }
1226    /*
1227     *    If accessing removable media, just print a generic error
1228     *    message and don't worry about error logging or specifics
1229     */
1230            if (doc_type[unit] == DT_FLOPPY) {
1231                cmn_err(CE_NOTE,"doc_: Floppy Access
                              Error: See Error Message");
1232                cmn_err(CE_CONT,"Section of the System
                              Administrator's Guide\n");
1233                goto berr;
1234            } else if (doc_type[unit] == DT_STREAMING) {
1235                cmn_err(CE_NOTE,"doc_: CTC Access Error:
                              See Error Message");
1236                cmn_err(CE_CONT,"Section of the System
                              Administrator's Guide\n");
1237                goto berr;
1238            }
1239    /*
1240     *    otherwise log the error and print a nasty message ...
1241     */
1242            if (prterr) {
1243                doc_elog[unit].diskdev = bp->b_dev
                              &  (IDNODEV|idslice((-1)));
```

Figure E–12 doc_intr Subordinate Driver Routine *(part 6 of 9)*

```
1244            /* The correct way to calculate the physical block
1245             * number is to simply read it back from the
1246             * controller so that defect mapping is accounted for.
1247             * Unfortunately, the controller apparently destroys
1248             * this field, so we just recalculate the number
1249             * assuming no defects.
1250             */
1251            if (idnodev(minor(bp->b_dev))){
1252                    /* access was "physical" */
1253                    doc_elog[unit].blkaddr = bp->b_blkno;
1254                    cmn_err(CE_WARN,"doc_: cannot
                                 access physical block %d",
1255                                 doc_elog[unit].blkaddr);
1256                    cmn_err(CE_CONT,"on controller %d,
                                drive %d: errcode 0x%x",
1257                                 contr(unit),
1258                                 subdev(unit),
1259                                 errcode);
1260            } else {
1261                    /* access was "logical" */
1262                    doc_elog[unit].blkaddr = bp->b_blkno
1263          + doc_vtoc[unit].v_part[idslice(minor(bp->b_dev))].p_start
1264           + doc_pdsect[unit].pdinfo.logicalst;
1265                    cmn_err(CE_WARN,"doc_: cannot access physical
                                block %d (lbn %d in partition %d)",
1266                                 doc_elog[unit].blkaddr,
1267                                 bp->b_blkno,
1268                                 idslice(minor(bp->b_dev)));
1269                    cmn_err(CE_CONT,"on controller %d,
                                drive %d: errcode 0x%x",
1270                                 contr(unit),
1271                                 subdev(unit),
1272                                 errcode);
1273            }
1274                    DEBUGhde("doc_int: bp->b_dev==0x%x,
                    bp->b_blkno==%d, doc_elog[%d].blkaddr==
                                 %d\n",bp->b_dev,bp->b_blkno,unit,
                    doc_elog[unit].blkaddr);
```

Figure E–12 doc_intr Subordinate Driver Routine (*part 7 of 9*)

```
1275                doc_elog[unit].readtype = HDECRC;
1276                doc_elog[unit].severity = HDEUNRD;
1277                doc_elog[unit].bitwidth = 0;
1278                doc_elog[unit].timestmp = time;
1279                for (i=0; i<12; i++)
1280                    doc_elog[unit].dskserno[i] =
1281                       doc_pdsect[unit].pdinfo.serial[i];

1282             /* do this last, because it may do more I/O
                            and cause more errors */
1283                hdelog(&doc_elog[unit]);
1284             }
1285    berr:
1286    /*
1287     *     mark the buffer in error
1288     */
1289             bp->b_flags |= B_ERROR;
1290             bp->b_error = EIO;
1291             goto err;
1292         }

1293    /*
1294     *    now update the residual, this makes EOF work
1295     */
1296       bp->b_resid -= doc_tcount[unit];
1297       doc_count[unit] += doc_tcount[unit];
1298    /*
1299     *    then if there is no more to transfer then go to the next buffer
1300     */
1301       if (bp->b_resid < doc_pdsect[unit].pdinfo.bytes) {
```

Figure E–12 doc_intr Subordinate Driver Routine (*part 8 of 9*)

```
1302    /*
1303     *    now unlink the buffer from the queue and set us up for the
1304     *        next io
1305     */
1306    err:
1307            dp->b_active = 0;
1308            dp->b_actf = bp->av_forw;
1309            dp->qcnt--;
1310             if (bp == (struct buf *)dp->acts)
1311                 dp->acts = (int)dp->b_actf;
1312    /*
1313     *    update status information
1314     */
1315            doc_time[unit].io_resp += lbolt - bp->b_start;
1316            doc_time[unit].io_act += lbolt - dp->io_start;
1317    /*
1318     *    wake up any processes waiting for this buffer
1319     */
1320            iodone(bp);
1321        }
1322        doc_retrys[unit] = 0;
1323    /*
1324     *    start the next io
1325     */
1326        doc_iostart(unit);
1327        DTRACE(" doc_intr: return\n");

1328    } /* end intr */
```

Figure E–12 doc_intr Subordinate Driver Routine (*part 9 of 9*)

```
1329    /*-------------------------------------------------------*/
1330    /*      Break up the request that came from physio into
1331     *      chunks of contiguous memory so we can get around
1332     *      the DMA controller limitations.  We must be sure
1333     *      to pass at least 512 bytes (one sector) at a
1334     *      time (except for the last request).
1335     */

1336    static
1337    doc_breakup(bp)
1338    register struct buf *bp;
1339    {
1340        dma_breakup(doc_strategy, bp);
1341    }
```

Figure E–13 doc_breakup Subordinate Routine

doc_read and doc_write Driver Entry Point Routines

The **read** and **write** entry point routines are very short and fairly simple. The **physck**(D3X) function checks that the requested block exists, then physio locks the block in memory (without moving it from user address space) and transfers the data. See Chapter 6, "Input/Output Operations," for a further discussion of physical I/O for a block-access device.

```
1342   /*-------------------------------------------------------*/
1343   /*
1344    *    physical read
1345    */
1346   doc_read(dev)
1347   {
1348       if (idnodev(minor(dev)) ||
              physck(doc_vtoc[iddn(minor(dev))].
                v_part[idslice(minor(dev))].p_size, B_READ))
1349            physio(doc_breakup, 0, dev, B_READ);
1350   }
```

Figure E–14 doc_read Entry Point Routine

```
1351   /*-------------------------------------------------------*/
1352   /*
1353    *    physical write
1354    */

1355   doc_write(dev)
1356   {
1357       if (idnodev(minor(dev)) ||
                physck(doc_vtoc[iddn(minor(dev))].
                 v_part[idslice(minor(dev))].p_size, B_WRITE))
1358           physio(doc_breakup, 0, dev, B_WRITE);
1359   }
```

Figure E–15 doc_write Entry Point Routine

doc_gocheck, doc_copy, and doc_setblk Subordinate Driver Routines

The **doc_gocheck** subordinate routine is called by the driver's initialization entry point routine. It uses four variables that are defined elsewhere

DOC_GOFLAG defined line 35, header file

GO_DONE defined line 58, header file

GOWAITSECS defined line 127, driver code

GOCHECKLPS defined line 128, driver code

```
1360   /*-------------------------------------------------------*/
1361   /*    gocheck -- if go flag is clear, return 0; if not :
1362    *    wait about GOWAITSECS secs, checking each loop;
1363    *   if it never clears return 1.
1364   /*-------------------------------------------------------*/

1365   static
1366   doc_gocheck(ctlr)
1367   int ctlr;                                  /* the doc_ board, 0-n */
1368   {
1369        int i;

1370        if(DOC_GOFLAG(ctlr) == GO_DONE) return 0;
                          /* quick exit on normal case */
1371        else {
1372                    for(i=(GOWAITSECS*GOCHECKLPS); i>0 ; i--)
1373                                if(DOC_GOFLAG(ctlr) == GO_DONE) return 0;
1374                    return 1;
1375        }
1376   } /* end doc_gocheck */
```

Figure E–16 doc_gocheck Subordinate Driver Routine

The **doc_copy** subordinate routine is called by the **doc_open** entry point routine to read physical description sector data, defect map, and the VTOC into a buffer when the device is first opened.

```
1378   /*
1379    *    copy count bytes by words
1380    */

1381   /*VARARGS*/
1382   static
1383   doc_copy(faddr, taddr, count)
1384   unsigned int *faddr;
1385   unsigned int *taddr;
1386   unsigned int count;
1387   {
1388       register unsigned int *fptr;
1389       register unsigned int *tptr;
1390       register int i,cnt;

1391       cnt = count/4;    /* # of words to transfer */
1392       tptr = taddr;
1393       fptr = faddr;
1394       for (i=0; i<cnt; i++)
1395           *tptr++ = *fptr++;
1396   }
```

Figure E–17 doc_copy Subordinate Driver Routine

The **doc_setblk** subordinate routine is used to setup the buffer for the **doc_copy** routine.

```
1397   /*-----------------------------------------------------------*/
1398   /*
1399    *    initialize buffer for command
1400    */

1401   /*VARARGS1*/
1402   static
1403   doc_setblk (bufhead, cmd, blkno, dev)
1404   struct buf *bufhead;
1405   u_char cmd;
1406   daddr_t blkno;
1407   dev_t dev;
1408   {
1409       clrbuf (bufhead);
1410       bufhead->b_flags |= cmd;
1411       bufhead->b_blkno = blkno;
1412       bufhead->b_dev = (dev | IDNODEV);
1413       bufhead->b_proc = 0x00;
1414       bufhead->b_flags &=  B_DONE;
1415   }
```

Figure E–18 doc_setblk Subordinate Driver Routine

doc_ioctl Driver Entry Point Routine

The **doc_** driver uses the **ioctl**(D2X) routine to format a disk subdevice. The **ioctl** routine is only available when the subdevice is accessed as a character device, not when it is mounted and accessed as a block device. Because it makes no sense to format a mounted disk device, this works perfectly well.

The I/O control commands in lines 1438 – 1441 are defined in lines 15 – 18 of the driver's header file. Other I/O control commands are defined in the *sys/vtoc.h* header file, to which all VTOC disk devices on the system must adhere. The relevant lines from *vtoc.h* are

```
/* driver ioctl() commands */
#define VIOC            ('V'<<8)
#define V_PREAD         (VIOC|1)    /* Physical Read */
#define V_PWRITE        (VIOC|2)    /* Physical Write */
#define V_PDREAD        (VIOC|3)    /* Read of Physical Description Area */
#define V_PDWRITE       (VIOC|4)    /* Write of Physical Description Area */
#define V_GETSSZ        (VIOC|5)    /* Get the sector size of media */
#define V_FORMAT        (VIOC|6)    /* Format disk */
#define V_GETFORMAT     (VIOC|7)    /* Get formatting parameters */
#define V_PDSETUP       (VIOC|8)    /* Set physical descriptors values */
                                    /* without writing them to disk */

/* ioctl() error return codes */

#define V_BADREAD       0x01
#define V_BADWRITE      0x02
#define V_BADFORMAT     0x04

/* Sanity word for the physical description area */
#define VALID_PD        0xCA5E600D
```

Figure E–19 Excerpt of sys/vtoc.h Header File

```
1416   /*-------------------------------------------------------*/
1417   /*
1418    *    Do device specific ioctls
1419    */

1420   /*ARGSUSED*/
1421   doc_ioctl(dev,cmd,argsptr,flag)
1422   char *argsptr;
1423   {
1424       struct buf *geteblk();
1425       struct buf *bufhead;
1426       int errno, xfersz;
1427       register int unit;
1428       unsigned int sector, mem, count, numbytes, defblock;
1429       struct pdsector *pd;
1430       struct io_arg arg, *args;
1431       int iplsave;            /* saved interrupt level */

1432       errno = DOC_NULL;
1433       args = &arg;
1434       unit = iddn(minor(dev));
1435       pd = &doc_pdsect[unit];

1436       DTRACE(" doc_ioctl: dev,cmd,f  %d,%d,%d\n",dev,cmd,flag);

1437       switch(cmd) {

1438       case IOCTL_DTRACEON: doc_dtrace = 1; break;
1439       case IOCTL_DTRACEOFF: doc_dtrace = 0; break;
1440       case IOCTL_DPRINTON: doc_dprint = 1; break;
1441       case IOCTL_DPRINTOFF: doc_dprint = 0; break;
```

Figure E–20 doc_ioctl Entry Point Routine (*part 1 of 13*)

```
1442    /*
1443     *    Format the media:   V_FORMAT is used to format
1444     *    a disk. The data structure vfmt_arg (defined
1445     *    in "sys/vtoc.h") is used to pass parameters.
1446     *
1447     *    N.B.
1448     *        The entire drive must be formatted in one shot.
1449     */
1450        case V_FORMAT: {
1451            register struct buf *bp;
1452            struct vfmt_arg vfmtarg, *format;
1453            register caddr_t cp;
1454            register u_short cyl;
1455            register u_char head;
1456            register int nsct;
1457            register char hard;
1458            register struct iobuf *dp;    /* pointer to queue header */
1459            DTRACE(" doc_ioctl: format option entered\n");
1460            format = &vfmtarg;
1461            if (copyin(argsptr, format, sizeof(struct vfmt_arg)) != 0) {
1462                u.u_error = EFAULT;
1463                return;
1464            }
1465            DPRINT(" doc_ioctl: format: r %d i %d t %d s %d\n",
1466                format->retval,format->interleave,
1467                format->trackcount,format->startsector);
1468            /* return fail unless asked to format entire disk */
1469            if (format->trackcount !=(pd->pdinfo.tracks*pd->pdinfo.cyls)) {
1470                errno = V_BADFORMAT;
1471                suword(&((struct io_arg *)argsptr)->retval,errno);
1472                DPRINT(" doc_ioctl: trackcount != pdinfo t * c\n");
1473                return;
1474            }
1475            dp = &doc_tab[contr(unit)];    /* Get the queue header */
1476    /*  ENTER CRITICAL REGION - spl5 = 10 on the
1477     *  processor = spl5 on the VMEbus
1478     */
1479            iplsave = spl5();
```

Figure E–20 doc_ioctl Entry Point Routine (*part 2 of 13*)

```
1480    /*  If there are no jobs on the controllers queue, and no
1481     *    other format in progress, grab the controller for a
1482     *    a format job.  Else sleep until iostart exhausts the
1483     *      queue and issues wakeup.
1484     */
1485
1486            while ((dp->b_actf != DOC_NULL)
1487                    || (doc_fmtflag[contr(unit)] != FMT_IDLE)) {
1488                sleep(dp,PZERO);
1489            }
1490            doc_fmtflag[contr(unit)] = FMT_INPROGRESS;

1491            /* do "format drive" command */
1492            /* error if go-flag says controller is busy */
1493                    if (doc_gocheck(contr(unit))) {
1494                                cmn_err(CE_WARN,
1495          "doc_ioctl: error: go-flag not clear before format\n");
1496                                cmn_err(CE_WARN,
1497          "doc_ioctl: aborting request\n");
1498                                return;
1499                    }
1500            /* set command */
1501            DOC_COMMAND(contr(unit)) = CMD_FORMAT | CMD_INTWD
1502                | ((doc_type[unit] == DT_HARD) ? CMD_HDIO : CMD_FLIO);

1503    #ifdef DRIVETMP
1504                    if (subdev(unit)==3) DOC_DRIVENO(contr(unit)) = 1 ;
1505                    else     DOC_DRIVENO(contr(unit)) = subdev(unit);
1506    #else
1507            DOC_DRIVENO(contr(unit)) = subdev(unit);
1508    #endif

1509            DPRINT(" doc_ioctl: 'format drive' unit %d type %d\n",
1510                unit,doc_type[unit]);
1511            DOC_GOFLAG(contr(unit)) = GO_START;
```

Figure E–20 doc_ioctl Entry Point Routine (*part 3 of 13*)

```
1512            /* sleep until interrupt routine wakes us */
1513            sleep(&doc_fmtflag[contr(unit)],PZERO);
1514            DPRINT(" doc_ioctl: back from sleep\n");
1515            if (doc_fmtflag[contr(unit)] == FMT_FAIL)
1516                {
1517                DPRINT(" doc_ioctl: format failed\n");
1518                u.u_error = EIO;
1519                }
1520            doc_fmtflag[contr(unit)] = FMT_IDLE;
1521            doc_iostart(unit);          /* let any pending io start */
1522    /*
1523     *    .FGIT CRITICAL REGION
1524     */
1525            splx (iplsave);
1526            break;
1527        }
1528    /*
1529     *     Physical Read
1530     */
1531        case V_PREAD:
1532            if (copyin(argsptr, args, sizeof(struct io_arg)) != 0) {
1533                u.u_error = EFAULT;
1534                return;
1535            }
1536            bufhead = geteblk();
1537            sector = args->sectst;
1538            mem = args->memaddr;
1539            count = args->datasz;
1540            DTRACE(" doc_ioctl: pread: %d bytes from
                          sector %d\n",count,sector);
1541            while (count)     {
1542                doc_setblk (bufhead, B_READ, sector, dev);
1543                bufhead->b_bcount = pd->pdinfo.bytes;
1544                doc_strategy(bufhead);
1545                iowait(bufhead);
1546                if (bufhead->b_flags & B_ERROR)     {
1547                    errno = V_BADREAD;
1548                    suword(&((struct io_arg *)argsptr)->retval,errno);
1549                    brelse(bufhead);
1550                    return;
1551                }
```

Figure E–20 doc_ioctl Entry Point Routine (*part 4 of 13*)

```
1552                xfersz = min(count, bufhead->b_bcount-bufhead->b_resid);
1553                if (copyout(bufhead->b_un.b_addr, mem, xfersz) != 0) {
1554                    u.u_error = EFAULT;
1555                    errno = V_BADREAD;
1556                    suword (&((struct io_arg *)argsptr)->retval,errno);
1557                    brelse(bufhead);
1558                    return;
1559                }
1560                if (!xfersz) break;
1561                sector += 1;
1562                count -= xfersz;
1563                mem += xfersz;
1564            }
1565            brelse(bufhead);
1566            break;
1567    /*
1568     *    Physical Write
1569     */
1570        case V_PWRITE:
1571            if (copyin(argsptr, args, sizeof(struct io_arg)) != 0) {
1572                u.u_error = EFAULT;
1573                return;
1574            }
1575            bufhead = geteblk();
1576            sector = args->sectst;
1577            mem = args->memaddr;
1578            count = args->datasz;
1579            DTRACE(" doc_ioctl: PWRITE sec %d count %d\n",sector,count);
1580            defblock = pd->pdinfo.defectst;
1581            numbytes = 0;
1582            while (count) {
1583                doc_setblk(bufhead, B_WRITE, sector, dev);
1584                bufhead->b_bcount = pd->pdinfo.bytes;
1585                xfersz = min(count, pd->pdinfo.bytes);
1586                if (copyin(mem, bufhead->b_un.b_addr, xfersz) != 0) {
1587                    u.u_error = EFAULT;
1588                    errno = V_BADWRITE;
1589                    suword(&((struct io_arg *)argsptr)->retval, errno);
1590                    brelse(bufhead);
1591                    return;
1592                }
```

Figure E–20 doc_ioctl Entry Point Routine (*part 5 of 13*)

```
1593                doc_strategy(bufhead);
1594                iowait(bufhead);
1595                if (bufhead->b_flags & B_ERROR)    {
1596                    errno = V_BADWRITE;
1597                    suword(&((struct io_arg *)argsptr)->retval, errno);
1598                    bufhead->b_bcount = pd->pdinfo.bytes;
1599                    brelse(bufhead);
1600                    return;
1601                }
1602            /* update memory image if special data */
1603                if (((bufhead->b_blkno == IDPDBLKNO) &&
                         (doc_type[unit] == DT_HARD)) ||
1604                  ((bufhead->b_blkno == IFPDBLKNO) &&
                         (doc_type[unit] == DT_FLOPPY)))
1605                {
1606                    doc_copy (bufhead->b_un.b_addr, pd, 512);
1607                    defblock = pd->pdinfo.defectst;
1608                }
1609                /* update defect map */
1610                if ((bufhead->b_blkno == defblock)
                          && (doc_type[unit] == DT_HARD)) {
1611                    defblock++;
1612                    doc_copy (bufhead->b_un.b_addr,
1613                        (((unsigned int) &doc_defect[unit]) +
1614                            numbytes),
1615                        512);
1616                    numbytes += 512;
1617                }
1618                /* update VTOC */
1619                if ((bufhead->b_blkno == (pd->pdinfo.logicalst+IDVTOCBLK))
1620                        && (doc_type[unit]==DT_HARD))
1621                    doc_copy(bufhead->b_un.b_addr, &doc_vtoc[unit],
1622                            sizeof(struct vtoc));
1623                sector += 1;
1624                count -= xfersz;
1625                mem += xfersz;
1626            }
1627            brelse(bufhead);
1628            break;
```

Figure E–20 doc_ioctl Entry Point Routine (*part 6 of 13*)

```
1629    /*
1630     *    Read the Physical Descriptor Sector off the disk
1631     */

1632        case V_PDREAD:
1633            DTRACE(" doc_ioctl: PDREAD\n");
1634            if (copyin(argsptr, args, sizeof(struct io_arg)) != 0) {
1635                u.u_error = EFAULT;
1636                return;
1637            }
1638            if (doc_type[unit] == DT_HARD) {
1639                bufhead = geteblk();
1640                doc_setblk (bufhead, B_READ, IDPDBLKNO, dev);
1641            }
1642            else if (doc_type[unit] == DT_FLOPPY) {
1643                bufhead = geteblk();
1644                doc_setblk (bufhead, B_READ, IFPDBLKNO, dev);
1645            }
1646            else break;
1647            bufhead->b_bcount = 512;
1648            doc_strategy(bufhead);
1649            iowait(bufhead);
1650            if (bufhead->b_flags & B_ERROR)    {
1651                errno = V_BADREAD;
1652                suword (&((struct io_arg *)argsptr)->retval,errno);
1653                brelse(bufhead);
1654                return;
1655            }
1656            if (copyout(bufhead->b_un.b_addr, args->memaddr,
                       sizeof(struct pdsector)) != 0) {
1657                u.u_error = EFAULT;
1658                errno = V_BADREAD;
1659                suword (&((struct io_arg *)argsptr)->retval,errno);
1660                brelse(bufhead);
1661                return;
1662            }
1663            brelse(bufhead);
1664            break;
```

Figure E–20 doc_ioctl Entry Point Routine (*part 7 of 13*)

```
1665    /*      Set up the controller with supplied pdsect values.
1666     *      Used to set up the parameters for a disk that has yet
1667     *      to be formatted and has no physical descriptor sector.
1668     *      Note that if the supplied pdsector is not valid, the
1669     *      current pdsector is copied in it's place and returned;
1670     *      nothing is initialized.
1671     */

1672        case V_PDSETUP: {
1673            struct pdsector pdtest;

1674            DTRACE(" doc_ioctl: PDSETUP\n");

1675            if (copyin(argsptr, args, sizeof(struct io_arg)) != 0) {
1676                u.u_error = EFAULT;
1677                return;
1678            }
1679            if (copyin(args->memaddr, &pdtest, sizeof(struct pdsector))!=0){
1680                u.u_error = EFAULT;
1681                errno = V_BADWRITE;
1682                suword (&((struct io_arg *)argsptr)->retval,errno);
1683                return;
1684            }
1685            if (pdtest.pdinfo.sanity != VALID_PD) {
1686                if (copyout(pd, args->memaddr, sizeof(struct pdsector))!=0) {
1687                    u.u_error = EFAULT;
1688                    errno = V_BADREAD;
1689                    suword (&((struct io_arg *)argsptr)->retval,errno);
1690                }
1691                return;
1692            }

1693    /*
1694     *      The pdsect for floppy disks is hard-wired into the driver
1695     *          It's not necessary to be able to change it
1696     */

1697            if (doc_type[unit] == DT_FLOPPY)
1698                return;
```

Figure E–20 doc_ioctl Entry Point Routine (*part 8 of 13*)

```
1699    /*
1700     *    Modify the drivers copy of the pdsect and then tell the
1701     *    controller about the new parameters.
1702     *
1703     *    The values coming in for tracks/cyl and sectors/track
1704     *    will be wrong if this is an attempt to set up "generic"
1705     *    values.  If so, adjust the values and recalculate the rest
1706     *    of the pdsect fields.
1707     */

1708              doc_pdsect[unit] = pdtest;
1709              DPRINT(" doc_ioctl PDSETUP: logicalst==%d
                        errlogst==%d defectst==%d\n",
1710                  doc_pdsect[unit].pdinfo.logicalst,
1711                  doc_pdsect[unit].pdinfo.errlogst,
1712                  doc_pdsect[unit].pdinfo.defectst);

1713     /* do "initialize drive" command, polling for completion */
1714              DPRINT(" doc_ioctl PDSETUP: 'init drive' on %d\n",unit);
1715     /* error if go-flag says controller is busy */
1716                      if (doc_gocheck(contr(unit))) {
1717                                      cmn_err(CE_WARN,
1718                      "doc_ioctl: error: go-flag not clear in PDSETUP\n");
1719                                      cmn_err(CE_WARN,
1720                      "doc_ioctl: aborting request\n");
1721                                      return;
1722                      }
1723              DOC_COMMAND(contr(unit)) = CMD_HDIO | CMD_INITDR;

1724     #ifdef DRIVETMP
1725                      if (subdev(unit)==3) DOC_DRIVENO(contr(unit)) = 1 ;
1726                      else     DOC_DRIVENO(contr(unit)) = subdev(unit);
1727     #else
1728              DOC_DRIVENO(contr(unit)) = subdev(unit);
1729     #endif
```

Figure E–20 doc_ioctl Entry Point Routine (*part 9 of 13*)

```
1730      DOC_NHEADS(contr(unit))=(u_char)(doc_pdsect[unit].pdinfo.tracks);
1731      DOC_MAXCYL(contr(unit))=(u_short)(doc_pdsect[unit].pdinfo.cyls-1);
1732      DOC_NSECTRK(contr(unit))=(u_char)(doc_pdsect[unit].pdinfo.sectors);
1733      DOC_NBYTSEC(contr(unit))=(u_short)(doc_pdsect[unit].pdinfo.bytes);
1734            DOC_GOFLAG(contr(unit)) = GO_START;
1735                     if(doc_gocheck(contr(unit))) {
1736                                 cmn_err(CE_WARN,
1737                           "doc_ioctl: goflag not clear after
                                  init drive in PDSETUP\n");
1738                                 return;
1739                     }
1740                     if(DOC_ERRCODE(contr(unit)) != ERR_NOERROR) {
1741                                 cmn_err(CE_WARN,
1742                           "doc_ioctl: PDSETUP reinit drive
                                  failed errcode==0x%x\n",
1743                                                DOC_ERRCODE(contr(unit)));
1744                                 return;
1745                     }
1746            break;
1747       }
1748   /*
1749    *    Write the supplied Physical Descriptor sector on to disk.
1750    */

1751       case V_PDWRITE:
1752           DTRACE(" doc_ ioctl PDWRITE\n");
1753           if (copyin(argsptr, args, sizeof(struct io_arg)) != 0) {
1754               u.u_error = EFAULT;
1755               return;
1756           }
1757           if (doc_type[unit] == DT_HARD) {
1758               bufhead = geteblk();
1759               doc_setblk (bufhead, B_WRITE, IDPDBLKNO, dev);
1760           }
1761           else if (doc_type[unit] == DT_FLOPPY) {
1762               bufhead = geteblk();
1763               doc_setblk (bufhead, B_WRITE, IFPDBLKNO, dev);
1764           }
```

Figure E−20 doc_ioctl Entry Point Routine (*part 10 of 13*)

```
1765            else break;
1766            bufhead->b_bcount = 512;
1767            if (copyin (args->memaddr, bufhead->b_un.b_addr,
1768                            sizeof(struct pdsector)) != 0) {
1769                u.u_error = EFAULT;
1770                errno = V_BADWRITE;
1771                suword(&((struct io_arg *)argsptr)->retval, errno);
1772                brelse(bufhead);
1773                return;
1774            }
1775            doc_strategy(bufhead);
1776            iowait(bufhead);
1777            if (bufhead->b_flags & B_ERROR)    {
1778                errno = V_BADWRITE;
1779                suword(&((struct io_arg *)argsptr)->retval, errno);
1780                brelse(bufhead);
1781                return;
1782            }
1783            brelse(bufhead);
1784            break;

1785    /*
1786     *    Return sector size for current disk
1787     */

1788        case V_GETSSZ:
1789            DTRACE(" doc_ ioctl GETSZ\n");
1790            if (copyin(argsptr, args, sizeof(struct io_arg)) != 0) {
1791                u.u_error = EFAULT;
1792                return;
1793            }
1794            suword(args->memaddr, pd->pdinfo.bytes);
1795            break;
```

Figure E–20 doc_ioctl Entry Point Routine (*part 11 of 13*)

```
1796    /*     Return sizes of interblock gaps and unformatted tracks
1797     *       and sectors.
1798     *         Used to determine what sectors to mark bad while
1799     *         setting up bad block tables.
1800     *         Uses formatarg data structure (defined
1801     *         in "sys/vtoc.h") to pass parameters.
1802     */

1803        case V_GETFORMAT: {
1804            struct trck_fmt formatarg, *formatargs;

1805            DTRACE(" doc_ ioctl GETFORMAT\n");

1806            formatargs = &formatarg;
1807            if (copyin(argsptr, formatargs,
                       sizeof(struct trck_fmt)) != 0) {
1808                u.u_error = EFAULT;
1809                return;
1810            }
```

Figure E–20 doc_ioctl Entry Point Routine (*part 12 of 13*)

```
1811    /*
1812     *     These parameters should be made less generic and determined
1813     *        according to device used
1814     *    These settings attempt to guarantee that any defect on the
1815     *    track will be caught, causing the entire track to be remapped.
1816     *    This is done because the actual format used by the controller
1817     *    is unknown.  Besides, it is most straightforward.
1818     */
1819    /* number of bytes in an unformatted ST506 track (I think) */

1820    #define RAWBPT    10416

1821            formatargs->bot_gap =  0;
1822            formatargs->eot_gap =  0;
1823            formatargs->sector_sz = RAWBPT/(pd->pdinfo.sectors);
1824            formatargs->track_sz = RAWBPT;

1825            if (copyout(formatargs, argsptr,
                      sizeof(struct trck_fmt))!=0) {
1826                u.u_error = EFAULT;
1827                return;
1828            }
1829            break;
1830        }

1831        default:
1832            u.u_error = EIO;
1833            break;
1834        }

1835        DTRACE(" doc_ioctl: return\n");
1836    }
```

Figure E–20 doc_ioctl Entry Point Routine (*part 13 of 13*)

Contents

Glossary

Introduction

This glossary is an alphabetical listing of terms and their definitions. The purpose of the glossary is to define specific system names, programming terms, and driver concepts for device driver writers.

In this glossary, notations are used for some entries to describe the location of the entry.

For structures, the definition gives the structure name followed by the header file in which the structure is defined. For example, `ccblock`(D4X) structure location is denoted in the glossary definition as: "Location: *tty.h*".

For flags, the definition gives the flag name followed by the associated structure and header file in which it is defined. For example, CARR_ON is a flag or value that is assigned to the structure member `tty` and its location is denoted in the glossary definition as:
"Location: **t_static-**`tty`-*tty.h*".

Any references to header files are found in the */usr/include/sys* directory. All references to source code are found in the */usr/src/uts/* computer (source code requires a special licensing agreement from AT&T). Consult the directory appropriate to the type of processor you are using.

NOTE: Source files have special reserve suffixes to denote the programming language in which the driver code is written. The *.c* denotes a file written in the C programming language. The *.s* denotes a file written in assembler language.

Terms and Definitions

ACP *See* Adjunct Communications Processor

ACU *See* automatic calling unit

Adjunct Data Processor
An adjunct data processing element that is housed in the ABUS cabinet and is plugged directly into the ABUS physical interface. The ADP containing a BIC, a WE® 32100 chip set running at 14 MHz, one SCSI port, and four megabytes of random access memory. The ADP provides computational and file service. *See also* Enhanced Adjunct Data Processor (EADP), Adjunct Communications Processor (ACP), and MP.

Adjunct Communications Processor (ACP)
An adjunct processing element that provides terminal support, networking connectivity, computational power, and printer interfaces for 3B4000 computer configurations. Unlike other adjuncts, the ACP is housed in a separate cabinet and connected to the appropriate ABUS slot by an XBI circuit board and XBUS cable.

ADP *See* Adjunct Communications Processor

AIC *See* alarm interface unit

alarm interface unit (AIC)
A UN-type circuit board that provides a series of alarm indications and the ability to access the computer from either the system console or a remote terminal. The AIC provides the following: external signaling of five alarm types, a sanity timer, non-volatile random access memory, a control and status register, and two RS-232C ports for the remote control feature.

alignment
The position in memory of a unit of data such as a word or half-word on an integral boundary. A data unit is properly aligned if its address is completely divisible by the data unit's size in characters. For example, a word is correctly aligned if its address is divisible by four. A half-word is aligned if its address is divisible by two.

allocated resource
A private map structure after memory has been allocated using the **malloc** command.

asm macro
The macro that defines a number of system functions used to improve driver execution speed. They are assembler language code sections (instead of C code). Location: `inline.h`.

asynchronous
An event occurring in an unpredictable fashion. A signal is an example of an asynchronous event. A signal can occur when something in the system fails, but it is not known when the failure will occur. This term is sometimes defined to be the interrupt level of driver.

automatic calling unit (ACU)
A device that permits processors to dial calls automatically over the communications network.

av_back
The `buf`(D4X) structure member that links the buffer to a free list. When no I/O transfer is currently scheduled, `buf` structures are linked together on an available list through the **av_forw** and **av_back** pointers. When a `buf` structure is needed for an I/O transfer, the first `buf` structure is taken from the available list. If no `buf` structures are available, the process needing a `buf` structure calls **sleep**, using the address of the head of the available list (`bfreelist`) as the *event* argument to **sleep**. Location: `buf`—*buf.h*

av_forw
The `buf`(D4X) structure member that links the buffer to a free list. When no I/O transfer is currently scheduled, a `buf` structure on the active I/O queue uses the **av_forw** pointer to maintain its place in the queue. The `buf` structures where no I/O transfer is currently scheduled are linked together on an available list via the **av_forw** and **av_back** pointers. When a `buf` structure is needed for an I/O transfer, the first `buf` structure is taken from the available list. If no `buf` structures are available, the process needing a `buf` structure calls **sleep**, using the address of the head of the list of available buffers (`bfreelist`). Location: `buf`—*buf.h*

awaken
The command that restarts a suspended process. Related commands are **untimeout**(D3X) and **wakeup**(D3X).

b_addr
The `buf`(D4X) structure member that contains the buffer's virtual address. Location: `buf`—*buf.h*

b_bcount
The `buf`(D4X) structure member that specifies the number of characters (bytes) to be transferred. Location: `buf`—*buf.h*

b_blkno
The `buf`(D4X) structure member that identifies which logical block on the device (defined by the minor device number) is to be accessed. Location: `buf`—*buf.h*

B_BUSY
The flag that indicates a buffer is in use. Location: **b_flags**—`buf`—*buf.h*

b_dev
The `buf`(D4X) structure member contains the major and minor device numbers of the device being accessed. Location: `buf`—*buf.h*

B_DONE
The flag that indicates the transfer has completed. Location: **b_flags**—`buf`—*buf.h*

b_error
The `buf`(D4X) structure member that holds the error code assigned by the kernel to the **u_error** member of the user data structure. This member is set with the B_ERROR flag. Location: `buf`—*buf.h*

B_ERROR
The flag that indicates an error occurred during an I/O transfer. Location: **b_flags**—`buf`—*buf.h*

b_flags
The `buf`(D4X) structure member that stores the status of the buffer and tells the driver whether the device is to be read from or written to. Location: `buf`—*buf.h*

B_PHYS
The flag that indicates the buffer is being used for physical (direct) I/O to a user data area. The **b_un** field contains the starting address for the user data. Location: **b_flags**—`buf`—*buf.h*

b_proc
The `buf`(D4X) structure member that contains the process table entry address for the process that is requesting a data transfer (when the transfer is unbuffered). This member is set to 0 (zero) when the

transfer is buffered. The process table entry performs proper virtual to physical address translation of the **b_un** member. Location: `buf`—*buf.h*

B_READ
The flag that indicates data is to be read from a peripheral device into main memory. Location: **b_flags**—`buf`—*buf.h*

b_resid
The `buf`(D4X) structure member that indicates the number of characters (bytes) not transferred because of an error. Location: `buf`—*buf.h*

b_start
The `buf`(D4X) structure member that holds the start time of the I/O operation. This member measures device response time. The system constant **lbolt** initiates this member. Location: `buf`—*buf.h*

b_un.b_addr
The `buf`(D4X) structure member that contains the virtual address of the buffer controlled by the buffer header. Data is written from this address to the device, or read to the address from the device. Location: `buf`—*buf.h*

B_WANTED
The flag that indicates the buffer is sought for allocation. Location: **b_flags**—`buf`—*buf.h*

B_WRITE
The flag that indicates the data is to be transferred from main memory to the peripheral device (the pseudo flag that occupies the same bit location as B_READ). This value does not exist, it can only be tested as the "not" state of B_READ. Location: **b_flags**—`buf`—*buf.h*

badrtcnt
The `hdedata`(D4X) structure member that indicates the number of unreadable tries made to a hard disk. Location: *hdelog.h*

base address
The address where a buffer is declared in memory. This can be a private map structure, or system buffers such as the `user` structure. In the latter case, the **u.u_base** member points to the base address of the `user` buffer.

base level
The code that synchronously interacts with a user program. The driver's initialization and switch table entry point routines constitute the base level. It is one of two logical parts of a driver. *See also* interrupt level.

BCI *See* block and character interface

bcopy(D3X)
The function that copies data between kernel addresses. This routine should never be used to copy data to or from an address in user space. Location: *ml/misc.s*

`bdevsw`(D4X)
The block driver switch table that is constructed during automatic configuration and exists only in memory or in the */unix* file (the structure is defined in *conf.h*).

`bfreelist`
The structure that points to a list of available (free) `buf` structures. The `bfreelist` address is used by processes accessing block devices as the *event* argument to **sleep**(D3X) when no free `buf` structures are available.

BIC *See* bus interface circuit

blkaddr
The `hdedata`(D4X) structure member that is a physical block address of a hard disk error in machine-dependent form. Location: *hdelog.h*

block
The basic unit of data for I/O access. A block is measured in bytes. The size of a block differs between computers, file system sizes, or devices.

block and character interface
A collection of driver routines, kernel functions, and data structures that provide a standard interface for writing UNIX System V, Release 3 block and character drivers.

block data transfer
The method of transferring data in units (blocks) between a block device such as a magnetic tape drive or disk drive and a user program.

block device
A device, such as a magnetic tape drive or disk drive that conveys data in blocks through the buffer management code (for example, the `buf` structure). *See also* character device.

block device switch table
The table constructed during automatic configuration that contains the address of each block driver base-level routine (**open**(D2X), **close**(D2X), **strategy**(D2X), and **print**(D2X)). This table is called `bdevsw` and its structure is defined in *conf.h*.

block driver
A driver for a device, such as a magnetic tape device or disk drive, that conveys data in blocks through the buffer management code (for example, the `buf` structure). One driver is written for each major number employed by block devices. On most systems, there are generally few block drivers.

block I/O
A data transfer method used by drivers for block access devices. Block I/O uses the system buffer cache as an intermediate data storage area between user memory and the device .

boot
The process of starting the operating system. The boot process consists of self-configuration and system initialization.

boot device
The boot device stores the boot code and necessary file systems to start the operating system.

bootable object file
A file that is created and used to build a new version of the operating system.

bootstrap
The process of bringing up the operating system by its own action. The first few instructions load the rest of the operating system into the computer.

brelse(D3X)
The function that releases unneeded buffers for block driver use. Location: *os/bio.c*

btoc(D3X)
The macro that converts bytes to clicks (pages). Location: *sysmacros.h*

`buf`(D4X)
The structure that provides buffering for block driver data transfers. Location: *buf.h*

buf.h
The header file that defines the `buf` structure. Location: *buf.h*

buffer
A staging area for input-output (I/O) processes where arbitrary-length transactions are collected into convenient units for system operations. A buffer consists of two parts: a memory array that contains data from the disk and a buffer header that identifies the buffer.

buffer_address
The `D_FILE`(D4X) structure member that contains the buffer address, which is set to (zero) before an **open** is called. Location: *system.h*

buffer_size
The `D_FILE`(D4X) structure member that sets the buffer size to NULL. Location: *system.h*

bus interface circuit (BIC)
A hardware interface between a bus and a processor. The BIC handles the sending and receiving of packets and distributed bus arbitration on the ABUS. A parallel interface connects each BIC to its processor.

BUSY
The flag that indicates output is in progress. Location: **t_state**—`tty`—*tty.h*

bzero(D3X)
The function that fills a buffer with zeros (clearing it) so that the buffer can be used for another purpose. Location: *ml/misc.s*

c_cc
The `clist` structure member that contains the number of characters in a `clist`. Location: `clist`—*tty.h*. Also, the `termio` structure member that contains the control characters contained in the `termio` structure. Location: `termio`—*termio.h*

c_cf
The `clist`(D4X) structure member that points to the first `cblock`. Location: `clist`—*tty.h*

c_cflag
The `termio` structure member that describes the terminal hardware control modes. **c_cflag** is represented in the `tty` structure by the **t_cflag** member. *See also* **termio**(7). Location: `termio`—*termio.h*

c_cl
The `clist`(D4X) structure member that points to the last `cblock`. Location: `clist`—*tty.h*

c_count
The `ccblock`(D4X) structure member that is initialized to the size of the `cblock` character array. This member is decreased by the number of characters in the `cblock` character buffer. The difference between **c_count** and **c_size** is used to indicate the number of characters in the buffer. Location: `ccblock`—*tty.h*

c_data
The `cblock` structure member that contains the data in the `cblock`. The maximum number of data characters in a `cblock` is defined by the CLSIZE constant. Location: `cblock`—*tty.h*

c_first
The `clist`(D4X) structure member that indexes the first character in the **c_data** array of a `cblock`. Location: `clist`—*tty.h*

c_flag
The `chead`(D4X) structure member that indicates a process is waiting for a `cblock`. Location: `chead`—*tty.h*

c_iflag
The `termio` structure member that describes the basic terminal input control modes. **c_iflag** is represented in the `tty` structure by the **t_iflag** member. *See also* **termio**(7). Location: `termio`—*termio.h*

c_last
The `cblock`(D4X) structure member that indexes to the last character in a **c_data** array of a `cblock`. Location: `cblock`—*tty.h*

c_lflag
The `termio` structure member used by the line discipline to control terminal functions. **c_lflag** is represented in the `tty` structure by the **t_lflag** member. *See also* **termio**(7). Location: `termio`—*termio.h*

c_line
The `termio` structure member that contains the line discipline value. The **t_line** member of the `tty` structure has the same purpose and value. Valid line discipline values are: 0, 1, and 2. The default standard value is 0. 1 is for a special protocol for AT&T 630 terminals and 2 is for use with **shl**(1), the shell **layers**(1) command. Location: `termio`—*termio.h*

c_next
The `cblock`(D4X) structure member that points to the next `cblock`. Location: `cblock`—*tty.h*

c_oflag
The `termio` structure member that specifies the system treatment of output. **c_oflag** is represented in the `tty` structure by the **t_oflag** member. *See also* **termio**(7). Location: `termio`—*termio.h*

c_ptr
The `ccblock`(D4X) structure member that points to the **c_data** character buffer. Location: `ccblock`—*tty.h*

c_size
The `chead`(D4X) structure member that indicates the size of the `cblock` character buffer. The **c_count** and **c_size** members are initialized to the size of the `cblock` character array (64 characters — CLSIZE). The **c_count** member is then decreased by the number of characters in the `cblock` character buffer. The difference between the two values indicates the number of characters in the buffer. Location: `chead`—*tty.h*

cache
A section of computer memory where the most recently used buffers, inodes, pages, and so on are stored for quick access. A separate controller is normally assigned to handle the cache I/O requests to leave the main processor free for other activity.

caddr_t
The character pointer data type used for memory addresses. Location: *types.h*

canon(D3X)
The function that transfers characters from **t_rawq** to **t_canq**. Location: *tty.c*

canonical processing
Terminal character processing in which the erase character, delete, and other commands are applied to the data received from a terminal before the data is sent to a receiving program. This type of processing can be thought of as "what the user really meant" when the data was keyed in at the terminal. Other terms used in this context are canonical queue, which is a buffer used to retain information while it is being canonically processed, and canonical mode, which is the state where canonical processing takes place. *See also* raw mode.

carrier
The continuous signal intermixed with another signal. The first (carrier) signal acts as a standard so that the second signal can be determined. The second signal is used for carrying data. A carrier is used by modems to convey data across phone lines. The modem indicates to the computer that the carrier is present by asserting the RS-232C received line signal detected signal lead to the computer. The 3B computers recognize the carrier signal when the carrier detect lead of the RS-232C interface is high.

CARR_ON
The flag that contains the signal software image indicating that a carrier is present for a terminal. Location: **t_state**—`tty`—*tty.h*

`cblock`(D4X)
The character block structure that contains a block of data used when a driver is accessing data from or to a terminal. Location: *tty.h*

`ccblock`(D4X)
The character control block structure that is used as a temporary buffer for characters not in a queue. Location: *tty.h*

`cdevsw`(D4X)
The character driver switch table is constructed during automatic configuration and exists in memory and in the */unix* file. Location: *conf.h.*

CE_CONT
The flag indicates that the message being passed to the **cmn_err** function should be displayed without a label such as NOTICE, PANIC, or WARNING. This display form appends the last message sent or displays an informative message not associated with an error. Location: *cmn_err.h*

CE_NOTE
The flag indicates that the message being passed to the **cmn_err** function should be displayed prefaced with "NOTICE:". Location: *cmn_err.h*

CE_PANIC
The flag indicates that the message being passed to the **cmn_err** function should be displayed prefaced with "PANIC:". Specifying CE_PANIC with **cmn_err** causes the computer to begin a panic. If a secondary panic state occurs while a panic message is being processed, the message is prefaced with "DOUBLE PANIC:". Location: *cmn_err.h*

CE_WARN
The flag indicates that the message being passed to the **cmn_err** function should be displayed prefaced with "WARNING:". Location: *cmn_err.h*

`cfreelist`(D4X)
The structure that contains a list of the free `cblocks`. `cfreelist` is declared to be a structure the same as `chead`. Location: *tty.h*

character device
The device, such as a terminal or printer that conveys data character by character. *See also* block device.

character driver
The driver that conveys data character by character between the device and the user program. Character drivers usually written for with terminals, printers, and network devices, although block devices such as tapes and disks also support character-access.

character I/O
The process of reading and writing to/from a terminal.

`chead`(D4X)
The structure indicates the start of the `cfreelist`. Location: *tty.h*

child process
When a process executes a **fork**(2) system call to create a new process, the new process is called a child process.

CLESC
The flag that indicates the last character processed was an escape character. Location: **t_state**—`tty`—*tty.h*

`clist`(D4X)
The structure that contains pointers to the first and last `cblocks`. A `clist` is used as a way of storing small quantities of data when a driver is moving data between a device controller and a terminal. Location: *tty.h*

close(D2X)
The base level routine that is used to end access to an open device. This routine is called only at the end of a device cycle and only if no other processes have the device open. The **close** routine examines the file table to ensure that the device is not being accessed, and then reinitializes the driver data structures and the device itself.

close(2)
The system call that releases a file descriptor when its use is no longer required.

clrbuf(D3X)
The function that is used by a block driver for zeroing a buffer in the `buf` structure. Location: *os/bio.c*

CLSIZE
The constant that specifies the number of data characters in a `cblock` is set by the CLSIZE constant. The current value for CLSIZE is 64. A single `cblock` can contain up to 64 characters. Location: *tty.h*

cmn_err(D3X)
The function that displays a message on the system console and stores the message in **putbuf**, or for causing the computer to panic. Location: *os/prf.c*

cmn_err.h
The header file that contains the four **cmn_err** severity-level definitions. These definitions define whether a message to be displayed on the system console does or does not cause a panic on the system. Location: *cmn_err.h*

common synchronous interface (CSI)
A set of functions designed to be used in drivers for virtual protocol machine (VPM) devices.

conf.h
The header file that contains the structure of the block device switch table (`bdevsw`), the character device switch table (`cdevsw`), and the line discipline switch table (`linesw`). Location: *conf.h*

control and status register (CSR)
Memory locations providing communication between the device and the driver. The driver sends control information to the to the CSR, and the device reports its current status to it.

controller
The circuit board that connects a device such as a terminal or disk drive to a computer. A controller converts software commands from a driver into hardware commands that the device understands. For example, on a disk drive, the controller accepts a request to read a file and converts the request into hardware commands to have the reading apparatus move to the precise location and send the information until a delimiter is reached.

copyin(D3X)
The function that copies data from a user program to a driver buffer. Location: *ml/misc.s*

copyout(D3X)
The function that copies data from a driver to user program space. Location: *ml/misc.s*

crash(1M)
A command that is used to analyze the core image.

CRC *See* cyclic redundancy check

critical code
A section of code is critical if execution of arbitrary interrupt handlers could result in consistency problems. The kernel raises the processor execution level to prevent interrupts during a critical code section.

CSI *See* common synchronous interface

CSR *See* control status register

ctob(D3X)
The macro that converts the clicks (pages) to bytes. Location: *sysmacros.h*

cyclic redundancy check (CRC)
A way to check the transfer of information over a channel. Binary code is sent over a channel in lengths. Each piece of code is divided by a fixed divisor. The result is added to the end of the message. When the message is received, the computer calculates the remainder and checks it against the transmitted remainder.

data structure
The memory storage area that holds dissimilar data types such as integers and strings. The data structures associated with drivers are used as buffers for holding data being moved between user data space and the device, as flags for indicating error device status, as pointers to link buffers together, and so on.

data terminal ready (DTR)
The signal that a terminal device sends to a host computer to indicate that a terminal is ready to receive data.

debug monitor (DEMON)
A low-level utility for verifying hardware and debugging software or firmware.

delay(D3X)
A function that is used by a block or character driver to delay the execution of a process for a specified time interval. Location: *os/clock.c*

demand paging
The implementation of demand paging allows processes to execute even though their entire virtual address space is not loaded in memory; so the virtual size of a process can exceed the amount of physical memory available in a system.

DEMON *See* debug monitor

device number
The value used by the operating system to designate a device. The device number contains the major number and the minor number. If it is denoted as internal, than the device number is logical and is known only to the kernel. External device numbers are half system-derived (the major number) and half created by the driver developer (the minor number).

dev_t
The C programming language data type declaration that is used to store the driver major and the minor device numbers. The data declaration is of the integer type **short**. Location: *types.h*

diagnostic
A software routine for testing, identifying, and isolating a hardware error. A message is generated to notify the tester of the results.

direct memory access controller (DMAC)
The WE32104/WE32204 chips that handle the access of data to and from memory, bypassing the CPU.

diskdev
The `hdedata`(D4X) structure member that contains the major/minor disk device number for the hard disk error. Location: *hdelog.h*

diskette.h
The header file for the 3B2 computer that contains structures and symbolic constants for floppy diskette access on the 3B2 computer. Location: *diskette.h*

dma_breakup(D3X)
The function that breaks up **physio** requests into manageable data blocks. Location: *physdsk.c*

DMAC *See* direct memory access controller

driver
The set of routines and data structures installed in the kernel that provide an interface between the kernel and a device. A driver provides all of the necessary programming so an interfaced device appears as a file to the rest of the UNIX operating system.

driver entry points
Driver routines that are activated during system initialization.

driver initialization
System initialization uses only the appropriate routines from the driver code and the information from the master file to initialize the drivers. Information such as the major/minor numbers that is so important when accessing driver switch table entry points is irrelevant when initializing a driver.

driver prefix
The unique two, three, or four digit prefix that is assigned in the driver master file and used as a prefix for driver routines.

driver routines
System structures and kernel functions used by the driver.

drv_rfile(D3X)
The 3B15 and 3B4000 computer function that reads a driver file. Location: *os/sys3.c*

drvinstall(1M)
The command that assigns the sequential major numbers file to the appropriate field in the master file.

dskserno
The `hdedata`(D4X) structure member that contains the disk pack serial number of the disk where the error is logged. Location: *hdelog.h*

DTR *See* data terminal ready

DUART dual universal asynchronous receiver transmitter. *See* universal asynchronous receiver transmitter

EADP *See* Enhanced Adjunct Data Processor

ECC *See* error correction code

EDT *See* equipped device table

EFAULT
The error message value that indicates a bad address. *See also* **intro**(2). Location: *errno.h*

EINTR
The error message value that indicates an interrupted system call. *See also* **intro**(2) in the *BCI Driver Reference Manual*. Location: *errno.h*

EINVAL
The error message value that indicates an invalid argument. *See also* **intro**(2). Location: *errno.h*

EIO *See* error in input/output

ELB *See* extended local bus

ELBU *See* extended local bus unit

Enhanced Adjunct Data Processor (EADP)
An adjunct processing element supporting two Small Computer System Interfaces (SCSI) (to two SCSI buses), eight or sixteen megabytes of memory, and a local BIC. Two EADPs may share a common peripheral.

enhanced ports (EPORTS)
EPORTS provides eight 8-pin modular jacks for serial RS-232C interface. EPORTS also includes software that must be installed before the hardware can be recognized by the system. The software contains diagnostic programs, enhanced ports driver, simple administration menus, and support files.

ENODEV
The error message value that indicates that there is no such device. *See also* **intro**(2) in the *BCI Driver Reference Manual*. Location: *errno.h*

EPERM
The error that indicates an attempt to modify a file forbidden except to its owner or superuser. It also returns for attempts by ordinary users to do things allowed only by the superuser. *See also* **intro**(2) in the *BCI Driver Reference Manual*. *Location: errno.h*

EQD_EFC
The error that indicates a device error for an external floppy controller. For further information, see the **hdeeqd**(D3X) function.

EQD_EHDC
The error that indicates a device error for an external hard disk controller. For further information, see the **hdeeqd** function.

EQD_ID
The error that indicates a device error for an integral disk drive. For further information, see the **hdeeqd** function.

EQD_IF
The error that indicates a device error for an integral floppy drive. For further information, see the **hdeeqd** function.

EQD_TAPE
The error that indicates a device error for a cartridge tape device. For further information, see the **hdeeqd** function.

equipped device table (EDT)
A list generated by the computer at boot time with an entry for each attached peripheral device. This list allows the computer to know what devices are active. *See* the *BCI Driver Development Guide*, Appendix A, *The Equipped Device Table (EDT)* for instructions on adding devices.

error correction code (ECC)
A generic term applied to coding schemes that allow for the correction of errors in one or more bits of a word of data. The error-correcting circuitry on an EADP/ADP provides single bit error detection and correction, an multiple bit error detection for RAM.

error in input/output (EIO)
An error that may occur on a call following the one to which it actually applied. This is a physical I/O error. *See also* **intro**(2). Location: *errno.h*

/etc/master.d
A directory that contains driver information files. The information supplies driver definitions and parameters used when a computer is configured. A master file is an individual file in this directory associated with a driver. Information in the master file is only used if there is a corresponding bootable object file in the */boot* directory.

/etc/system
A file that contains statements indicating whether a driver should be included or excluded during configuration.

extended local bus (ELB)
An extension to the local bus providing additional I/O slots.

extended local bus unit (ELBU)
A 3B4000 computer Master Processor or 3B15 computer card cage for UN-type circuit boards that provides local bus I/O slots in addition to those in the basic control unit and the growth control unit.

external major numbers
External major numbers for software devices are **static** and are assigned sequentially to the appropriate field in the master file by the **drvinstall**(1M) command; external major numbers for hardware drivers correspond to the board slot and are dynamically assigned by the **lboot** process as system boot time.

external minor number
Part of the name of the device file usually corresponds to the unit number of the device to be accessed via the file, or specifically, the minor number.

EXTPROC
The flag that indicates a peripheral is performing semantic processing of data. Semantic processing entails input validation of the characters received from a character device. Location: **t_state**—`tty`—*tty.h*

FAPPEND
The flag that indicates a file is open. This value is passed to the driver **open**(D2X) routine by the kernel. Location: *file.h*

FCREAT
The constant that opens a new file. This value is passed to the driver **open** routine by the kernel. Location: *file.h*

FEXCL
The constant that causes an **open**(D2X) to fail if a file already exists if used with FCREAT. This value is passed to the driver **open** routine by the kernel. Location: *file.h*

file.h
The header file that contains definitions used for opening and accessing a file. Location: *file.h*

file_name
The `D_FILE`(D4X) structure member that contains the name of the file to be accessed. Location: *system.h*

file service
The use of an EADP/ADP and MP for file system storage and manipulation.

firmware
Computer circuitry, such as silicon chips, that contains commands that can be read, but not deleted. Firmware, also known as read-only memory (ROM), generally contains commands that are used to boot the operating system.

firmware.h
The header file that contains pointers to a computer's firmware. Some of these pointers include random access memory start addresses, structures for system generation, booting, error handling, and for sending pumpcode to an intelligent controller. Location: *firmware.h*

FNDELAY(D2X)
The constant that indicates non-blocking I/O permission has been granted to a user program for file access. This value is passed to the driver **open**(D2X) routine by the kernel. Location: *file.h*

FREAD(D2X)
The constant that indicates read permission has been granted to a user program for file access. This value is passed to the driver **open**(D2X) routine by the kernel. Location: *file.h*

FSYNC(D2X)
The constant that indicates synchronous write permission is granted to a user program for file access. This value is passed to the driver **open**(D2X) routine by the kernel. Location: *file.h*

FTRUNC(D2X)
The constant that opens an existing file and truncates its length to zero. This value is passed to the driver **open** routine by the kernel. Location: *file.h*

fubyte(D3X)
The function that copies a character (byte) from user program space to a driver. This is an obsolete function. Location: *ml/misc.s*

fuword(D3X)
The function that copies a word of data from user program space to a driver. This is an obsolete function. Location: *ml/misc.s*

FWRITE
The constant that indicates write permission has been granted to a user program for file access. This value is passed to the driver **open**(D2X) routine by the kernel. Location: *file.h*

getc(D3X)
The function that gets a character from a `clist`. Location: *io/clist.c*

getcb(D3X)
The function that gets the first `cblock` on a `clist`. Location: *io/clist.c*

getcf(D3X)
The function that gets a free `cblock`. Location: *io/clist.c*

geteblk(D3X)
The function that gets an empty block. Location: *os/bio.c*

getmajor(1M)
The command that returns the major number for the specified device.

getsrama(D3X)
The function that gets the starting address of the segment descriptor table (SDT). It is used on the 3B15 computer and the 3B4000 MP to access the proper memory management unit (MMU) when doing direct memory access (DMA). Location: *immu.h*

getsramb(D3X)
The function that gets the length of segment descriptor table (SDT). It is used on the 3B15 computer and the 3B4000 MP to access the proper memory management unit (MMU) when doing direct memory access (DMA). Location: *immu.h*

getvec(D3X)
The function for the 3B2 computer that gets an interrupt vector given a virtual board address. Location: *os/machdep.c*

header file
A file that ties declarations together for a set of programs. It guarantees all source files are supplied with the same definitions and declarations.

hdeeqd(D3X)
The function that initiates hard disk error logging. Location: *io/hde.c*

hdelog(D3X)
The function that logs hard disk errors to a table in the kernel and to the console. Location: *io/hde.c*

high water mark
The point at which data being processed in the output `clists` is transmitted to the terminal.

IASLP
The flag that indicates the processes associated with the device should be awakened when input completes. Location: **t_state**—`tty`—*tty.h*

IDFC *See* integral disk file controller

IDUART integral dual universal asynchronous receiver transmitter. *See* universal asynchronous receiver transmitter

init(D2X)
The routine that initializes a device. **init** is called by the operating system when the computer is started.

initialization entry points
Driver initialization routines that are executed during system initialization. *See also* **init** and **start**.

input/output accelerator (IOA)
A UN-type circuit board that directs peripheral controllers to interface with the 3B15 computer or 3B4000 Master Processor local bus and main memory.

int(D2X)
The routine processes a device interrupt. The driver interrupt handler is entered when a hardware interrupt is received from a driver-controlled device.

integral disk file controller (IDFC)
A UN-type circuit board that interfaces to a storage module device controller (SMDC), which interfaces FSD disk drives to the 3B4000 Master Processor or the 3B15 computer. The IDFC resides in an I/O slot on the primary local bus.

interface
The routines, data structures, command arguments, major and minor numbers, and master and system files used to develop a driver.

internal major numbers
An index into the switch tables. Internal major numbers are assigned by the self-configuration process when the drivers are loaded, and probably change every time the system is booted.

internal minor numbers
The internal minor number is assigned by the driver writer (although there are conventions enforced for some types of devices by some utilities), and usually refers to subdevices of the device.

interprocess communication (IPC)
A set of facilities supported through software that enables independent processes, running at the same time, to exchange information through messages, semaphores, or shared memory.

interrupt entry points
Driver interrupt routines that are activated when an interrupt is received from a hardware device. The system accesses the interrupt vector table, determines the major number of the device, and passes control to the appropriate interrupt routine.

interrupt priority level (IPL)
The interrupt priority level (1 to 15) at which the device requests that the CPU call an interrupt process. This priority can be overridden in the driver's **int** routine for critical sections of code with the **spl***n*(D3X) function.

interrupt vector
Interrupts from a device are sent to the device's interrupt vector, activating the *interrupt entry point* for the device.

IOA *See* input/output accelerator

ioctl(D2X)
The character driver base level routine that conveys hardware or software control information to a character device.

iodone(D3X)
The function used by a block driver for resuming the execution of a process after a block I/O request has completed. Location: *os/bio.c*

iomove(D3X)
A function used for copying data. The routine decides whether the source and target addresses are within kernel or user program space and calls **bcopy**(D3X), **copyin**(D3X), or **copyout**(D3X) accordingly. This is an obsolete function. Location: *os/move.c*

iowait(D3X)
The function used by a block driver for suspending execution of a process until a request for input or output completes. Location: *os/bio.c*

IPC *See* interprocess communication

IPL *See* interrupt priority level

ISOPEN
The flag that indicates a device is open. Location: **t_state**—`tty`—*tty.h*

ivec *See* interrupt vector

kernel buffer cache
A linked list of buffers used to minimize the number of times a block-type device must be accessed.

kseg(D3X)
The function that makes memory pages available for a driver's use. Location: *os/mmgt.c*

l_close
The `linesw`(D4X) structure member that invokes the **ttclose**(D3X) function (for line discipline zero) to discontinue access to a terminal. Location: `linesw`—*conf.h*

l_input
The `linesw`(D4X) structure member that invokes the **ttin** function (for line discipline zero) to service an input interrupt from a terminal. Location: `linesw`—*conf.h*

l_ioctl
The `linesw`(D4X) structure member that invokes the **ttioctl**(D3X) function (for line discipline zero) to service an **ioctl** request for a terminal. Location: `linesw`—*conf.h*

l_mdmint
The `linesw`(D4X) structure member handles modem interrupts. In line discipline zero, this member is set to **nulldev** and is non-functional. Location: `linesw`—*conf.h*

l_open
The `linesw`(D4X) structure member that invokes the **ttopen**(D3X) function (for line discipline zero) to service an open request for a terminal. Location: `linesw`—*conf.h*

l_output
The `linesw`(D4X) structure member that invokes the **ttout**(D3X) function (for line discipline zero) to service an output interrupt for a terminal. Location: `linesw`—*conf.h*

l_read
The `linesw`(D4X) structure member that invokes the **ttread**(D3X) function (for line discipline zero) to service a read request from a terminal. Location: `linesw`—*conf.h*

l_write
The `linesw`(D4X) structure member that invokes the **ttwrite**(D3X) function (for line discipline zero) to service a write request to a terminal. Location: `linesw`—*conf.h*

layers(1)
The UNIX system user command that provides multiple command windows on a terminal.

LBE *See* local bus extender

lbolt
The system variable of **time_t** type that contains the number of Hertz (HZ) clock ticks since system boot time. It can be used to determine a precise relative time. For example, a driver can determine

the elapsed time for an I/O operation by taking the difference between the recorded starting time **lbolt** value and the completion time **lbolt** value.

lboot
The **lboot** program runs when the system is booted and reads the #VEC field in the driver's master file to determine the number of interrupt vectors per controller and assigns numbers accordingly.

line discipline switch table
Line discipline interprets input and output characters between the operating system and a terminal. The line discipline switch table, `linesw`(D4X), is a list of pointers to the character driver processing kernel routines that interpret and buffer the characters received from and sent to a terminal. The `linesw` structure is defined in */usr/include/sys/conf.h*. The protocols for processing and buffering characters are referred to as a line discipline. Valid line discipline values are: 0, 1, and 2. Line discipline 0 is the default standard value, 1 is for a special protocol for AT&T 630 terminals, and 2 is for use with **shl**(1), the shell **layers**(1) command. The line discipline switch table is defined in *conf.h* header file. For further information, see the *BCI Driver Development Guide*, Chapter 7, "Drivers in the TTY Subsystem."

line discipline zero
See line discipline switch table.

`linesw`(D4X)
See line discipline switch table.

local bus extender (LBE)
A circuit board that provides the interface between the 3B4000 Master Processor or the 3B15 computer and the bus extension facilities. The LBE is optional, but if purchased, it must be located in the basic control unit of the basic cabinet.

logical controller numbers
Numbers that are assigned sequentially by the central controller firmware at self-configuration time.

logmsg(D3X)
The function that logs an error message. Location: *errlog.c*

logstray(D3X)
The function that logs spurious (nonlocatable) errors and interrupts. Location: *io/errlog.c*

longjmp(D3X)
The function that transfers program control from the current point of execution back to a previous point quickly. Location: *ml/cswitch.s*

low water mark
The point at which more data is requested from a terminal because the amount of data being processed in the character lists has fallen creating room for more.

MAJOR table
The MAJOR table maps internal major numbers to the external major number. Each table is a character array that is 128 entries long.

major(D3X)
The macro that obtains an internal major device number from a device number. Location: *sysmacros.h*

major number
The number that identifies a device class. Internal major numbers are known only to the kernel and are logical values. The `bdevsw` and `cdevsw` switch tables are referenced by the internal major number. External major numbers are found in two ways. If the major number is associated with a hardware device, the number is created when the computer is automatically configured and accessed with the **getmajor**(1M) command, If the major number is associated with a software driver, the number is created by **drvinstall**(1M).

makedev(D3X)
The macro that creates an external device number from a major number and a minor number. Location: *sysmacros.h*

malloc(D3X)
The function that allocates a private `map` structure. Location: *os/malloc.c*

manufacturer's defect table (MDT)
A disk defect table supplied by the manufacturer of a given disk.

map.h
The header file that is used when declaring private `map` structures. The header file provides the definition of the **mapinit** function. Location: *map.h*

mapinit(D3X)
The macro that initializes a private space management map. Location: *map.h*

mapwant(D3X)
The macro that requests a free buffer for a private space management map. Location: *map.h*

master file
The file that supplies information to the system initialization software to describe the attributes of a driver. This file also contains the driver prefix and device number, and whether it is a software or hardware driver.

Master Processor (MP)
The controlling processor that interfaces with the adjuncts on the ABUS thru the XBUS connection and a remote BIC. The MP contains a WE 32100 chip set running at 14 MHz, and 8 or 16 megabytes of random access memory. The MP is the single point of control for bootstrap, system configuration, centralized resource service, and maintenance.

max(D3X)
The function that returns the larger of two numbers. Location: *ml/misc.s*

MDT *See* manufacturer's defect table

member
A field or element of a structure.

memory management
The memory management scheme of the UNIX operating system imposes certain restrictions on drivers that transfer data between devices.

memory management unit (MMU)
WE 32101 and WE 32201 chips provide support for running the paging scheme of memory management. The chips make use of tables maintained by the kernel for performing address translations.

mfree(D3X)
The function that frees a space in private memory. Location: *os/malloc.c*

min(D3X)
The function that returns the smaller of two numbers. Location: *ml/misc.s*

MINOR table
The table that maps internal minor numbers to the external major number. Each table is a character array that is 128 entries long.

minor(D3X)
The macro that obtains an internal minor device number from a device number. Location: *sysmacros.h*

minor device number
A number used to identify a specific device on a controller. An internal minor number is known only to the kernel and is a logical number. An external minor number is created by the driver developer and is usually a collection of information about the device.

mknod(1M)
The command that creates special device files or nodes that are used by the system to access the device.

MMU *See* memory management unit

modem
A contraction of modulator-demodulator. A modulator converts digital signals from the computer into tones that can be transmitted across phone lines. A demodulator converts the tones received from the phone lines into digital signals so that the computer can process the data.

MP *See* Master Processor

multiprocessor
Multiprocessor architecture contains two or more CPUs that share common memory and peripherals. A multiprocessing computer can provide greater throughput, because processes can run concurrently on different processors.

NCC
The constant that indicates the maximum number of control characters defined in the **t_cc** member of `tty` structure (in *tty.h*). The valid control characters are described in **termio**(7) and contained in the **c_cc** array of the `termio` structure. The default value for NCC is 8. Location: *termio.h*

nodev(D3X)
The function that indicates that a driver base-level routine was omitted. **nodev** places the ENODEV error message in **u.u_error** when **nodev** is called. When the `cdevsw` and `bdevsw` switch tables are built, the kernel interrogates each driver to determine the names of the base level routines. A character driver normally has five base-level routines: **open**(D2X), **close**(D2X), **read**(D2X), **write**(D2X), and **ioctl**(D2X). A block driver normally has four base-level routines: **open**, **close**, **strategy**(D2X), and **print**(D2X). When one of the base-level routines does not exist in the driver, the kernel substitutes **nodev** in the routine's position in the switch table. Location: *os/subr.c*

NULL
The constant that indicates a 0 (zero). Location: *param.h*

OASLP
The flag that indicates the processes associated with the device should be awakened when output completes. Location: **t_state**—`tty`—*tty.h*

open(D2X)
The driver switch table entry point routine that is called by the system when a user program invokes the **open**(2) instruction. The kernel then executes the driver's **open** routine.

open_close
The `D_FILE`(D4X) structure member that sets an open or close flag. Location: *system.h*

open.h
The header file that contains constants specifying a driver **open** routine. Location: *open.h*

OPOST
The flag that indicates output characters are post-processed as indicated by the other flags in the same structure. Location: *termio.h*

otyp
The argument used in the **open**(D2X) a routine. The possible values for **otyp** are described in *open.h*. Location: *system.h*

page descriptor (PD)
The base address of a memory page used by the memory management unit (MMU) to map pages within paged segments from virtual to physical memory.

page descriptor table (PDT)
A table containing a list of page descriptors (PDs) used by the memory management unit (MMU) to map pages within paged segments from virtual to physical memory.

p_pgrp
The `proc`(D4X) structure member that contains the process group identification number. The number is used to determine which processes should receive a HANGUP or BREAK signal. A driver detects these signals. Location: `proc`—*proc.h*

p_pid
The `proc`(D4X) structure member that contains the process identification number. Location: `proc`—*proc.h*

p_pri
The `proc`(D4X) structure member that contains the priority of a process. The value is used by the scheduler to determine which process gets to execute from a number of executable processes. Location: `proc`—*proc.h*

p_uid
The real user ID of a process. Location: `chead`—*tty.h*

panic
The state where an unrecoverable error has occurred. In most cases, when a panic occurs, a message is displayed on the console to indicate the cause of the problem. The computer must be rebooted or repaired to remedy the problem.

param.h
The header file that contains definitions for constants that change infrequently. Examples of such constants are HZ, NULL, and PZERO. Location: *param.h*

parent process
Almost every process is created when another process executes a **fork**(2) system call. This process is called the parent process. The newly created process is called the child process.

PCATCH
The constant that instructs the kernel **sleep**(D3X) routine not to call the kernel **longjmp** routine, but to return value 1 to the calling routine. Location: *param.h*

PCB *See* process control block

PD *See* page descriptor

PDI *See* portable driver interface

PDT *See* page descriptor table

physck(D3X)
The function that verifies a requested block exists on the device. Location: *os/physio.c*

physio(D3X)
The function that processes an I/O request. Location: *os/physio.c*

PIR *See* programmed interrupt requests

portable driver interface (PDI)
A collection of driver routines, kernel functions, and data structures that provide a standard interface for writing UNIX System V block drivers. PDI is usable on all 3B2, 3B15, and 3B4000 computers running UNIX System V Release, 2.0.5, 3.0, 3.1, or later.

prefix
A two-, three-, or four-character name that uniquely identifies a driver's routines to the kernel. The prefix name starts each routine in a block or character driver. For example, a RAM disk might be given the **ramd** prefix. If it is a block driver, the routines are **ramdopen**, **ramdclose**, **ramdstrategy**, and **ramdprint**. The *prefix* must be registered with AT&T.

print(D2X)
The routine that uses the minor number to determine what part of the device is not performing correctly.

proc(D2X)
The routine that processes various character device-dependent operations. This routine is required for a character driver that accesses the `tty` or `linesw` structures.

`proc`(D4X)
The structure that contains information required by the operating system for a process
Location: *proc.h*

process
An instance of a program in execution.

process control block (PCB)
An operating system structure that stores process information.

process ID (PID)
The kernel identifies each process by its ID.

proc.h
The header file contains the `proc` structure used only by the kernel for storing information about the currently running process. Location: *proc.h*

programmed interrupt request (PIR)
An interrupt sent by a software device.

psignal(D3X)
The function that sends a signal to a single process. Location: *os/sig.c*

pumpcode
Executable code that is downloaded to the controller.

putc(D3X)
The function that places a character on a `clist`. Location: *io/clist.c*

putcb(D3X)
The function that links a `cblock` to a `clist`. Location: *io/clist.c*

putcf(D3X)
The function that places a `cblock` on the free list. Location: *io/clist.c*

putbuf
A buffer, accessible with **crash**(1M), that records messages displayed with **cmn_err**(D3X). A message is placed in **putbuf** routinely each time **cmn_err** is called, or exclusively, if an exclamation mark (!) is encoded in the first position of the message. **putbuf** can be avoided by encoding a caret (^) in the first position of the message.

PZERO
The constant that indicates the point in the range of **sleep**(D3X) priority values that determines whether the system will awaken a sleeping process on receipt of a signal. PZERO is generally set to 25. Priority values with a range of 0 to PZERO, keep the system from awakening sleeping processes receiving a signal. Priority values with a range of PZERO+1 to 39 cause the system to awaken a sleeping process when a signal is received. When a sleeping process is awakened on a signal, the process is awakened before the event on which it was sleeping occurs. Location: *param.h*

raw I/O
Movement of data directly between user address spaces and the device. Raw I/O is used primarily for administrative functions where the speed of a specific operation is more important than overall system performance.

raw mode
The method of transmitting data from a terminal to a user without processing. This mode is defined in the line discipline modules. *See also* canonical processing.

rcvint
A member of the `sysinfo`(D4X) structure. It increments the entry to **rint**(D2X). Location: `sysinfo`—*sysinfo.h*

read(D2X)
The routine for the `cdevsw`(D4X) table that copies information from a character device to a user address space.

read(2)
The system call that reads data from a file. It is only used in user programs and not in a driver.

readtype
The `hdedata`(D4X) structure member that indicates either a CRC or ECC hard disk error. Location: *hdelog.h*

remote file sharing (RFS)
Transparent sharing of directory structures by independent machines.

RFS *See* remote file sharing

rint(D2X)
The routine that services a receive interrupt. A receive interrupt occurs when a device has data ready to be read.

routine
A section of C programming language or assembler code handling a specific task. Driver routines differ from a complete program or other types of routines because driver routines do not include the syntax required to identify a program to the system. In the C programming language, a program is identified by the use of the **main()** function. A driver routine does not contain **main()**.

RTO
The flag that indicates a timeout is in progress for a device operating in raw mode. Location: **t_state**—`tty`—*tty.h*

SCCS *See* Source Code Control System

SCSI *See* Small Computer System Interface

SCSI driver interface (SDI)
A collection of machine-independent input/output controls, functions, and data structures, that provide a standard interface for writing SCSI target drivers to access a SCSI device.

SCSI local interface circuit (SLIC)
A UN-type circuit board that provides the interface between two Small Computer System Interface buses and the primary local bus on the 3B4000 Master Processor or the 3B15 computer.

SD *See* segment descriptor

SDI *See* SCSI driver interface

SDT *See* segment descriptor table

SGS *See* Software Generation System

segment descriptor (SD)
The base address of a paged segment that is used by the memory management unit (MMU) to map contiguous segments from virtual to physical memory.

segment descriptor table (SDT)
A table of segment descriptors (SDs) used by the memory management unit (MMU) to map contiguous segments from virtual to physical memory.

self-configuration
Self-configuration refers to the construction of the specific kernel for the computer. Because drivers function as part of the kernel, you need to create or modify self-configuration files and reconfigure the system to install your driver.

semantic processing
Semantic processing entails input validation of the characters received from a character device.

severity
The `hdedata`(D4X) structure member that indicates hard disk error severity; an error is either marginal or unreadable. Location: *hdelog.h*

shl(1)
The system user command lets a user have multiple simultaneous shell command line prompts (called layers). On terminals equipped with multiple windowing capability (such as the Teletype 4425), after a number of windows are created, **shl** allows a user to be able to execute shell commands from each window. **shl** is terminal independent. Each window (layer) is given a unique process ID.

signal(D3X)
The function that sends a signal to a process group. Location: *os/sig.c*

signal.h
The header file contains signal values described in the **signal**(2) system call. Location: *signal.h*

single board computer (SBC)
The WE 321SB single board computer (SBC). A computer on a single circuit board that permits installable device drivers.

sleep(D3X)
The function that suspends the execution of a process until an event occurs. **sleep** is normally given the address of a structure as its argument. This structure may be a repository for data from an I/O request. When an I/O request completes, the driver checks for processes that have called **sleep** with the address of the structure. The **wakeup**(D3X) routine is called by the driver to awaken the sleeping processes. Location: *os/slp.c*

SLIC *See* SCSI local interface circuit

Small Computer System Interface (SCSI)
In the 3B4000 or 3B15 computer, SCSI refers to the disk and tape interface supported by the SCSI local interface circuit (SLIC) and an EADP/ADP or ACP. *See also* SCSI controller, SCSI device, SCSI host adapter, SCSI local interface circuit (SLIC), and SCSI peripheral cabinet.

Software Generation System (SGS)
A package of tools designed to aid in program development.

Source Code Control System (SCCS)
A utility for tracking, maintaining, and controlling access to source code files.

special device file
The file that identifies the device's access type (block or character), the external major and minor numbers of the device, the device name used by user-level programs, and security control (owner, group, and access permissions) for the device.

spl*(D3X)
A series of functions used to suppress or restore the interrupt level for the execution of critical code. **spl1, spl4, spl5, spl6, spl7, splhi, splpp,** and **spltty** suppress some or all interrupts so that critical code can be executed without the danger of having an interrupt disrupt execution. **spl0** restores the state where all interrupts are serviced. **splx** returns the interrupt state to a previous state. Location: *ml/misc.s*

splhi(D3X)
The function that ensures interrupts do not occur while critical regions of code are executing. **splhi** blocks all interrupts. Location: *ml/misc.s*

splx
The function that restores the previous interrupt inhibit level. For example, if a previous **spl4** call was made, and then **splhi** was called, the driver program should return to the **spl4** state. **splx** is used to ensure that the correct level is reached. Location: *ml/misc.s*

sptalloc(D3X)
The function that allocates pages of memory. Location: *os/page.c*

sptfree(D3X)
The function that frees previously allocated pages of memory. Location: *os/page.c*

start(D2X)
A system initialization driver entry point routine.

strategy(D2X)
The block driver routine that transmits data between the buffer cache and the device. One of the functions of the **strategy** routine is to schedule reads and writes for maximum device efficiency. For example, on a hard disk, the heads take a certain amount of time to move in and out to access data. The **strategy** routine may group read and write requests together by the relative head position that each request is calling, while the disk heads are moving back for a new movement command to be issued by the disk controller. When the disk heads are ready, the read and write requests are given to the controller, and sorted by the data's position on the disk relative to how the disk head moves. The heads are then allowed to move in a coordinated way allowing the data to be read and written in the most efficient manner. In addition to scheduling, **strategy** may validate the block number contained in the read or write request, and also check the device for the end-of-file condition.

STREAMS
A modular system used to build device drivers and protocol handlers that reside in the kernel. STREAMS allow modules to pass messages to implement a full-duplex connection between the kernel and the device.

subyte(D3X)
The function that copies a character (byte) from a driver to user program space. This is an obsolete function. Location: *ml/misc.s*

suser(D3X)
The function checks to see if the current process has superuser permissions. Location: *os/fio.c*

suword(D3X)
The function that copies a word of data from a driver to user program space. This is an obsolete function. Location: *ml/misc.s*

switch table
The operating system that has two switch tables, `cdevsw`(D4X) and `bdevsw`(D4X). These tables hold the entry point routines for character and block drivers and are activated by I/O system calls.

switch table entry points
Driver routines that are activated through `bdevsw` or `cdevsw` switch tables.

sxt driver
The shell layers **shl**(1) device driver.

synchronous
Events occurring at fixed, regular, or predictable intervals.

synchronous device
A device that communicates with the CPU in a fixed, regular, or predictable way.

sysadm(1M)
The system administrative command that contains menus for performing many operations and administrative tasks.

`sysinfo`(D4X)
The structure used by character drivers **rint**(D2X) and **xint**(D2X) driver interrupt routines to indicate the number of times each routine is entered. Location: *sysinfo.h*

system initialization
The routines from the driver code and the information from the master file to initialize that initialize the system (including device drivers).

T_BLOCK
The constant that indicates that the driver **proc**(D2X) routine should block further input because the input queue has reached the high water mark. T_BLOCK turns off TTXON and turns on TTXOFF and TBLOCK in the **t_state** member of the `tty` structure (in the driver **proc** routine). Location: *tty.h*

T_BREAK
The constant that indicates that the driver **proc**(D2X) routine should send a break character to a terminal device. The driver sets the **t_state** member of the `tty` structure to TIMEOUT and initiates delay timing. Refer to the **proc** routine in Appendix D for an example of how T_BREAK is used. Location: *tty.h*

t_canq
The `tty`(D4X) structure member that contains data accepted from a terminal after canonical processing (erase character, deletes, and so on) has taken place. Location: `tty`—*tty.h*

t_cc
The `tty`(D4X) structure member that contains an array of control characters. Location: `tty`—*tty.h*

t_cflag
The `tty`(D4X) structure member that corresponds to the control modes flag (**c_cflag**) defined in the **termio** structure. *See also* **termio**(7). Location: `tty`—*tty.h*

t_delct
The `tty`(D4X) structure member used by the `tty` subsystem to keep track of the number of delimiters found while performing semantic processing of data from a terminal. Semantic processing entails input validation of the characters received from a character device. Location: `tty`—*tty.h*

T_DISCONNECT
The constant that indicates that the driver **proc**(D2X) routine should disconnect a `tty` device. Location: *tty.h*

t_iflag
The `tty`(D4X) structure member that corresponds to the input modes **c_iflag** defined in the **termio** structure and described in **termio**(7). Location: `tty`—*tty.h*

T_INPUT
The constant that indicates the driver **proc**(D2X) routine should flag a terminal device to receive input. Location: *tty.h*

t_lflag
The `tty`(D4X) structure member that corresponds to the local modes **c_lflag** defined in the **termio** structure. *See also* **termio**(7).
Location: `tty`—*tty.h*

t_line
The `tty`(D4X) structure member that holds the line discipline type specified in the **c_line** member of the `termio` structure. Refer to **termio**(7) for more information.

t_oflag
The `tty`(D4X) structure member that corresponds to the output modes **c_oflag** defined in the `termio` structure. *See also* **termio**(7). Location: `tty`—*tty.h*

T_OUTPUT
The constant that indicates the driver **proc**(D2X) routine should initiate output to the terminal device. This condition is not set if the device is busy or if output has been suspended. Location: *tty.h*

t_outq
The `tty`(D4X) structure member that contains all of the data that is accepted from a terminal.
Location: `tty`—*tty.h*

t_pgrp
The `tty`(D4X) structure member that identifies the process group associated with the device. This member is needed to send signals to the process group. Location: `tty`—*tty.h*

t_proc
The `tty`(D4X) structure member that holds the address of a character driver **proc** routine.
Location: `tty`—*tty.h*

t_rawq
The `tty`(D4X) structure member that contains the data being sent to a terminal. Location: `tty`—*tty.h*

t_rbuf
The `tty`(D4X) structure member that is the receive buffer for a TTY device. Location: `tty`—*tty.h*

T_RESUME
The constant that indicates the driver **proc**(D2X) routine should resume output on a terminal because a CTRL-q character has been received. The TTSTOP bit in the **t_state** member of the `tty` structure should be cleared. Location: *tty.h*

T_RFLUSH
This constant is the same as T_UNBLOCK if TBLOCK is set in the **t_state** member of the `tty` structure; otherwise, this indicator means nothing. Location: *tty.h*

t_state
The `tty`(D4X) structure member that maintains the internal state of the device and the driver. Note the **t_state** member is fully utilized and cannot be extended for additional state information that a particular driver may need. Location: `tty`—*tty.h*

T_SUSPEND
The constant that indicates that the driver **proc**(D2X) routine should suspend output to a terminal because a CTRL-s character has been received. The TTSTOP bit in the **t_state** member of the `tty` structure should be set. Location: *tty.h*

t_tbuf
The `tty`(D4X) structure member is the transmit buffer for a TTY device. Location: `tty`—*tty.h*

T_TIME
The constant that indicates the driver **proc**(D2X) routine should delay timing because a BREAK, carriage return, and so on, has completed. Location: *tty.h*

T_UNBLOCK
The constant that indicates the driver **proc**(D2X) routine should allows more input because the input queue has gone below the high-water mark. The driver **proc** routine resets TTXOFF and TBLOCK in the **t_state** member of the `tty` structure. Location: *tty.h*

T_WFLUSH
The constant that indicates the driver **proc**(D2X) routine should clear out the characters in the transmit buffer. Location: *tty.h*

TACT
The flag that indicates a timeout is in progress for a TTY device. Location: **t_state**—`tty`—*tty.h*

TBLOCK
The flag that indicates the driver has sent a control character to the terminal to block transmission from the terminal. Location: **t_state**—`tty`—*tty.h*

TCFLSH
The constant that flushes the input or output queue for a TTY device. It is used by **ttiocom**(D3X) and is described in the *Administrator's Reference Manual* under **termio**(7). Location: *termio.h*

TCGETA
The constant that gets and stores the parameters for a terminal. (This constant is used by **ttiocom** and is described in the *Administrator's Reference Manual* under **termio**(7).) Location: *termio.h*

TCSBRK
This constant is used as a **case** condition in the **ttiocom** function. When an **ioct**(2) system call accesses TCSBRK, **ttiocom** calls **ttywait**(D3X) to allow the UART to drain. If the argument to the **ioctl** command is zero, the driver **proc**(D2X) routine is called with the T_BREAK argument to send a break character to the device and to initiate delay timing. If the **ioctl** argument is other than zero and after the **proc** routine completes, control returns to the caller. Location: *termio.h*

TCSETA
The constant that sets parameters for a terminal from a structure. This constant is used by **ttiocom** and is described in the *Administrator's Reference Manual* under **termio**(7). Location: *termio.h*

TCSETAW
This constant is a **case** condition in the **ttiocom** function that is used to wait for output to drain from a UART and to flush the read and write buffers before new parameters are set. Location: *termio.h*

TCXONC
The constant that suspends output or restarts suspended output. This constant is used by **ttiocom** and is described in the *Administrator's Reference Manual* under **termio**(7). Location: *termio.h*

termio.h
The header file that contains information relevant to accessing a TTY device. Location: *termio.h*

TIMEOUT
The flag that indicates a delay timeout is in progress. Location: **t_state**—`tty`—*tty.h*

timeout(D3X)
The function that suspends the execution of a process for a designated time interval. Location: *os/clock.c*

timestmp
The `hdedata`(D4X) structure member that puts a time stamp on a hard disk error logging table entry. Location: *hdelog.h*

trace(7)
A special file that allows event records generated within the kernel to be passed to a user program so that the activity of a driver or other system routines can be monitored for debugging purposes.

ttclose(D3X)
The function that closes a TTY device. Location: *io/tt1.c*

ttin(D3X)
The function that moves a character from the **t_rbuf** to the raw queue. Location: *io/tt1.c*

ttinit(D3X)
The function that initializes a `tty` structure. Location: *io/tty.c*

ttiocom(D3X)
The function that examines the parameters of a TTY device. Location: *io/tty.c*

ttioctl(D3X)
The function that changes the parameters of a TTY device. Location: *io/tt1.c*

TTIOW
The flag that indicates the process associated with the device is sleeping, awaiting completion of output to the terminal. Location: **t_state**—`tty`—*tty.h*

ttopen(D3X)
The function that opens a TTY device. Location: *io/tt1.c*

ttout(D3X)
The function that moves a TTY character output queue to **t_tbuf**. Location: *io/tt1.c*

ttread(D3X)
The function that processes an input TTY character. Location: *io/tt1.c*

ttrstrt(D3X)
The function that restarts TTY output after a delay timeout. Location: *io/tt1.c*

tttimeo(D3X)
The function that times a character device terminal read request. Location: *tt1.c*

ttwrite(D3X)
The function that moves a TTY character user data space to the **t_outq** device. Location: *io/tt1.c*

TTSTOP
The flag that indicates output has been stopped by a CTRL-s character received from the terminal. Location: **t_state**—`tty`—*tty.h*

TTXOFF
The flag that indicates the CPU has hit the high water mark in receiving data from a TTY device. Calls the driver **proc** routine with T_BLOCK as the *cmd* argument. Location: **t_state**—`tty`—*tty.h*

TTXON
The flag that indicates the data processed by the CPU has hit the low-water mark. Calls the driver **proc** routine with T_UNBLOCK as the *cmd* argument. Location: **t_state**—`tty`—*tty.h*

ttxput(D3X)
The function that puts characters into the TTY output buffer (**t_outq**). Location: *tt1.c*

`tty`(D4X)
The structure that maintains all information relevant to a TTY device. Location: *tty.h*.

tty.h
The header file that contains a structure used for buffering data between a terminal device and a character driver. Location: *tty.h*

ttyflush(D3X)
The function that clears the I/O queues used in a character driver. Location: *io/tty.c*

TTYHOG
The constant that defines the maximum number of characters allowed in a TTY device's raw queue. Location: *tty.h*

ttywait(D3X)
The function that delays a process until an I/O operation has completed. Location: *io/tty.c*

types.h
The header file that contains data type definitions for expressions frequently used in the kernel and drivers. Location: *types.h*

u.u_base
The `user`(D4X) structure member that specifies the base address for I/O actions to and from user data space. Location: `user`—*user.h*

u.u_count
The `user` structure member that specifies the number of characters (bytes) not yet transferred during an I/O transaction. Location: `user`—*user.h*

u.u_error
The `user` structure member that returns an error code to the user (in the **errno** external variable). Valid error codes are described in **intro**(2), Chapter 4 of the *BCI Driver Development Guide*. Location: `user`—*user.h*

u.u_gid
The `user` structure member that contains the effective group identification number. This member provides a process with the access permissions group. Location: `user`—*user.h*

u.u_offset
The `user` structure member that specifies the offset into the file where data is being transferred to or from. Location: `user`—*user.h*

u.u_procp
The `user` structure member that contains the address of the `proc`(D4X) structure associated with the user process. Location: `user`—*user.h*

u.u_qsav
The `user` structure member that is an argument to the kernel **longjmp**(D3X) routine. This address is set automatically by the operating system each time a driver is started. Location: `user`—*user.h*

u.u_rgid
The `user` structure member that identifies the real group ID. Location: `user`—*user.h*

u.u_ruid
The `user` structure member that identifies the real user ID. Location: `user`—*user.h*

u.u_segflg
The `user` structure member is an flag that determines if the user kernel initiated the I/O. Location: `user`—*user.h*

u.u_ttyp
The `user` structure member that contains the address of the process group member (**t_pgrp**) of the `tty` structure for the terminal associated with this process. Location: `user`—*user.h*

u.u_uid
The `user` structure member that contains the effective user ID. This member provides access permissions of another user. Location: `user`—*user.h*

UART *See* universal asynchronous receiver transmitter

universal asynchronous receiver transmitter (UART)
A circuit board chip that conveys bytes of data between a serial communications line and a microprocessor (for example between a 3B computer and a TTY device). In transmit mode, the UART reads a byte from a microprocessor's data bus and outputs the byte a bit at a time on a serial line for a terminal. In receive mode, the UART converts bit data from a serial line and forms a byte which is then given to the microprocessor. UARTs can generally handle data speeds between 50 bits per second (bps) and 19.2 thousand bps with character widths from 5 to 8 bits.

unkseg(D3X)
The function that frees previously allocated memory pages. Location: *os/page.c*

untimeout(D3X)
The function that cancels a previous **timeout**(D3X) call. Location: *os/clock.c*

user.h
The header file that contains the `user`(D4X) structure. Location: *user.h*

`user`(D4X)
The structure that contains status information for a process. One `user` structure is defined for each process in the kernel. The kernel uses the information for process status checking. For the currently running process, **u** is used to access the members of the user block. Location: *user.h*

useracc(D3X)
The function that verifies a **user data space**
The portion of kernel memory used to store data for programs executing in user space.

user space
The part of the operating system where programs that do not have direct access to the kernel structures and services execute. The UNIX operating system is divided into two major areas: the user program and the kernel. Drivers execute in the kernel, and the user programs that interact with drivers generally execute in the user program area. This space is also referred to as user data area.

useracc(D3X)
The function that verifies a user has access to a requested data structure. Location: *os/probe.c*

virtual protocol machine (VPM)
A software module that handles communications to the IOA.

volume table of contents (VTOC)
Lists the beginning and ending points of the disk partitions by the system administrator for a given disk.

VPM *See* virtual protocol machine

VTOC *See* volume table of contents

vtop(D3X)
The function that converts a virtual address to a physical address. Location: *ml/misc.s*

wakeup(D3X)
The function that resumes execution of a suspended process. Location: *os/sop.c*

WOPEN
The flag that indicates the driver is waiting for an open request to complete.
Location: **t_state**—`tty`—*tty.h*

write(2)
The system call that stores information on a device. Information is copied from user program space to a driver. This function is executed only from a user program and not from a driver.

write(D2X)
The routine for the `bdevsw`(D4X) or `cdevsw`(D4X) tables that conveys data from user space to kernel space.

xint(D2X)
A routine that services a transmit interrupt.

xmtint
The `sysinfo`(D4X) structure member that increments the entry to **xint**.
Location: `sysinfo`—*sysinfo.h*

Index

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

AT&T values your opinion. We'd like to know how well this document meets your needs. Please check the appropriate column below to indicate your opinion of the document for the categories listed on the right.

If we need more information may we contact you? Yes ☐ No ☐

Name (Optional) _____

Job Title or Function _____

Organization _____

Address _____

_____

_____

Phone ( ) _____

Does the document meet your needs?
Why or why not? _____

	Excellent	Good	Fair	Poor
Ease of Use				
Accuracy				
Examples				
Completeness				
Organization				
Appearance				
Writing				
Clarity				
Illustrations				

_____

Please make at least one comment. _____

_____

_____

BCI Driver Development Guide, Issue 1 **307-191**

AT&T values your opinion. We'd like to know how well this document meets your needs. Please check the appropriate column below to indicate your opinion of the document for the categories listed on the right.

If we need more information may we contact you? Yes ☐ No ☐

Name (Optional) _____

Job Title or Function _____

Organization _____

Address _____

_____

_____

Phone ( ) _____

Does the document meet your needs?
Why or why not? _____

	Excellent	Good	Fair	Poor
Ease of Use				
Accuracy				
Examples				
Completeness				
Organization				
Appearance				
Writing				
Clarity				
Illustrations				

_____

Please make at least one comment. _____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 5 NEW PROVIDENCE N.J.

POSTAGE WILL BE PAID BY ADDRESSEE

AT&T
4513 Western Avenue
Lisle, Illinois 60532
Attn: District Manager—Documentation

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 5 NEW PROVIDENCE N.J.

POSTAGE WILL BE PAID BY ADDRESSEE

AT&T
4513 Western Avenue
Lisle, Illinois 60532
Attn: District Manager—Documentation