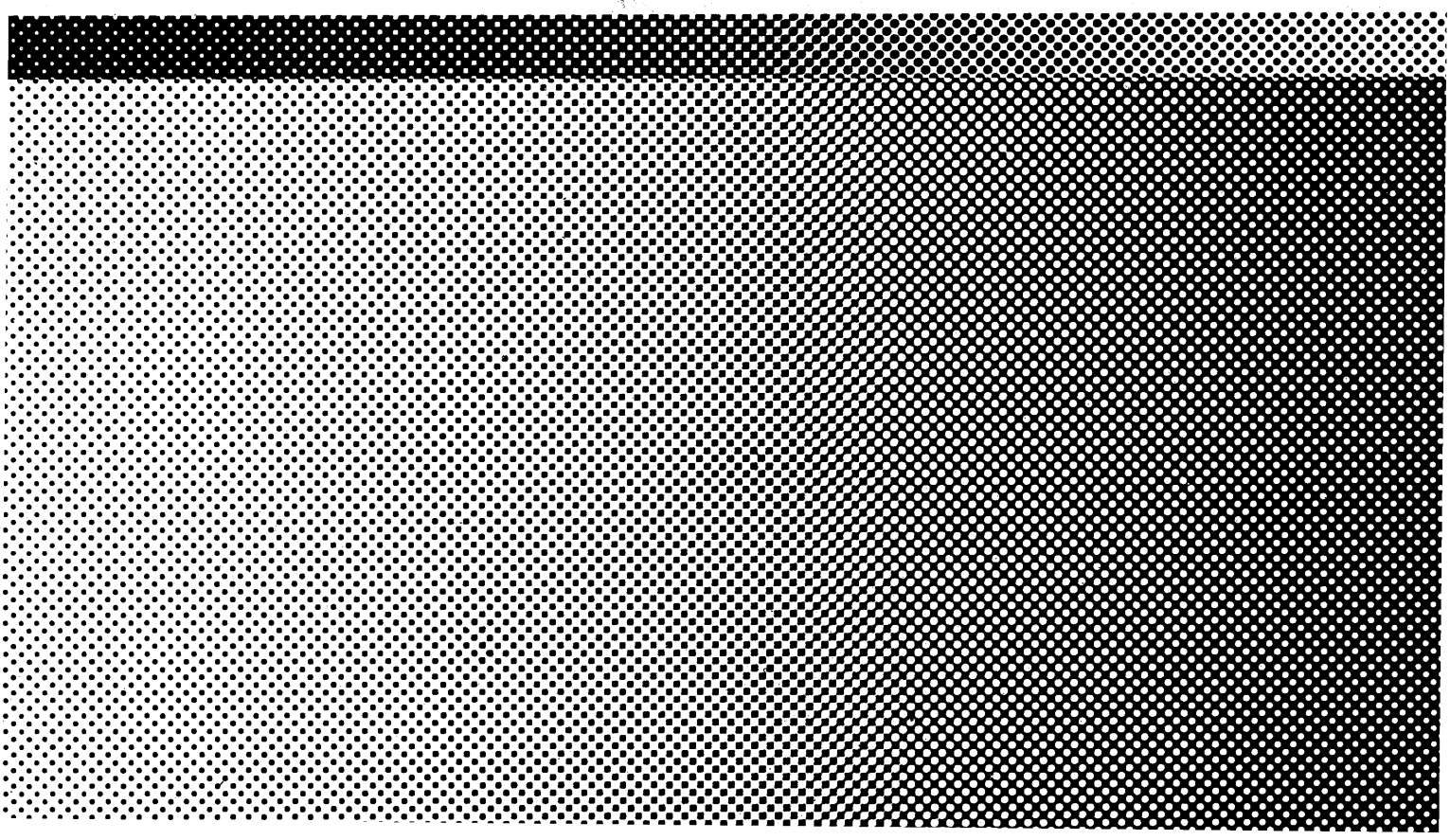
UNIX® System V, Release 3
Block and Character Interface (BCI) Driver Reference Manual

307-192
Issue 2
Preliminary

UNIX® System V, Release 3
Block and Character Interface (BCI) Driver Reference Manual



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 307-192

Contents

System Maintenance Functions (D8X)

Glossary

Chapter 1: About This Document

Contents

About This Document

The *AT&T Block and Character Interface (BCI) Driver Reference Manual* (shortened hereafter to *BCI Driver Reference Manual*) provides manual pages a driver designer uses to write, install, and debug drivers in the UNIX® System V, Release 3, environment. The manual defines entry point routines that must be written, the kernel functions that should be used, the data structures with which the BCI drivers interact, and the standard library functions used to write a diagnostics file. It is a companion to the *AT&T Block and Character Interface (BCI) Driver Development Guide* (shortened hereafter to *BCI Driver Development Guide*), which 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).

Please note that code samples in the *BCI Driver Reference Manual* are code fragments and are not intended to be copied and compiled into drivers.

For more information about this document, see the "How to Use This Document" section in this chapter.

Driver Development Series

The *BCI Driver Reference Manual* is part of the *AT&T Driver Development Series*. The *Block/Character Interface (BCI) Driver Development Guide* is a companion document to this manual. Other documents being developed for 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 3B4000 Adjunct Communications Processor (ACP), Adjunct Data Processor (ADP), and Enhanced Adjunct Data Processor (EADP). Information that is applicable to only certain adjuncts is so marked.

Purpose

The *BCI Driver Reference Manual* provides the manual pages a driver writer needs to write, install, and debug device drivers in the UNIX System V environment.

Intended Audience

Both this book and the *BCI Driver Development Guide* 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

The *BCI Driver Reference Manual*, the most closely related document to the *BCI Driver Development Guide*, is divided into four, alphabetically-arranged sections that provide specific information 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). (Section D8X includes only a small subset of all of the system board firmware functions. More functions will be documented in later versions of this document.)

A *Glossary* is also included at the end of this book.

Notes on Section D3X

Certain kernel functions in Section D3X are designated "OBSOLETE" in this book (Table 1-1). These functions are provided for reference, but may not be supported in future system releases.

Table 1–1 OBSOLETE Functions

Function	Reason	Replaced By
fubyte, fuword	Error return code is indistinguishable from data	**copyin**
iomove	Requires driver use of **u.u_segflg** and adds an extra layer to I/O transfers	**bcopy, copyin, copyout**
subyte, suword	Error return code is indistinguishable from data	**copyout**

Section D3X does not include pages for the **panic** and **printf** kernel functions, which were used in previous UNIX System V releases. AT&T no longer supports the use of these functions because they

have been replaced by the **cmn_err** function. NOTE: The kernel **printf** function is different from **printf**(3S). **printf**(3S) is supported.

Organization of Driver Reference Manuals

Driver reference manual pages are referenced much like the *UNIX System V Reference Manual* pages: the page name is followed by a section number in parentheses. All driver reference manual sections begin with a "D" to distinguish them as driver reference pages.

Currently, the reference pages for the different interfaces are published in separate volumes. Each manual contains four sections for the specific interface:

D2 Driver Routines
D3 Kernel Functions used by drivers
D4 System Data Structures accessed by drivers
D8 System Maintenance Functions

Each section number is suffixed with a letter indicating the interface covered. The suffixes used are:

X Block and Character Interface (BCI)
P Portable Device Interface (PDI)
I SCSI Device Interface (SDI)

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* (this document). 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 D2X of the this book. 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 D2X of the *UNIX System V Programmer's Reference Manual.*

The STREAMS interface for writing character drivers is currently documented in books outside the Driver Development Series. See the list of "Related Documents" at the end of this chapter.

Conventions Used in This Document

Table 1-2 lists the textual conventions used in this book. These conventions are also used in the *BCI Driver Development Guide*.

Table 1–2 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**

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 adjunct processor number. For example:

> */etc/master.d* directory means:
> on a uniprocessing system: */etc/master.d*
> on the 3B4000 computer: */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 1–3 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 Reference Manual.* Its ordering code number is 307-192.

Related Documents

Driver Development

UNIX System V Block/Character Interface (BCI) Driver Development Guide (307-191)
discusses driver development concepts, debugging, performance, installation, and other related driver topics.

UNIX System V Portable Driver Interface (PDI) Driver Design Reference Manual (305-014)
defines the kernel routines, functions, 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 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 variou releases of the UNIX System V operating system, including an explanation of how drivers relate to the rest of the kernel.

UNIX System V Reference Manuals (see the table following for ordering numbers)
the standard reference materials for the UNIX 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 1-4 gives the select codes for the UNIX System V reference manuals that are published for each AT&T computer covered in this documentation.

Table 1–4 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

Section D2X: Driver Routines(D2X)

Contents

Introduction

Section D2X describes the system entry-point[1] routines a driver developer uses to create a driver plus the **proc** subroutine that is required for TTY drivers. All reference pages for driver routines have the (D2X) cross reference code.

Each driver is organized into two parts, the base level and the interrupt level. The base level interacts with the kernel and the user program; the interrupt level interacts with the device.

Each driver has a prefix that is defined in its master file. This prefix is prepended to the routine name to form the name of the actual routine in the driver. So, for a driver with the "pre_" prefix, the driver code may contain routines named **pre_open**, **pre_close**, **pre_init**, **pre_int**, and so forth.

Driver routines can call subroutines that are assigned names by the driver writer. Subroutines should be type **static**, in which case no rules apply for naming subroutines. However, the prefix should be used in subroutine names to increase code readability.

Because subroutines are variable, planning, writing, and execution of these routines is the responsibility of the developer.

In this section, reference pages contain the following headings:

- NAME summarizes the routine's purpose
- SYNOPSIS describes the routine's entry point in the source code
- ARGUMENTS describes arguments used to invoke the routine
- DESCRIPTION provides general information about the routine
- DEPENDENCIES lists possible dependent routine conditions
- SEE ALSO gives sources for further information

1. Drivers with system entry-point routines are called from the switch tables (bdevsw and cdevsw), when the computer is started, and when a device generates an interrupt.

Overview of Driver Routines

Table D2X-1 lists the driver routines presented in this section. See individual reference pages in this section for further information.

Table D2X−1 Driver Routine Types

<table>
<tr><th colspan="4">Base Level Routine Types</th></tr>
<tr><th colspan="2">System Defined Name Routines:</th><th colspan="2">Subordinate Driver Routines</th></tr>
<tr><td>Initialization Routines

Form:
prefix init( )
prefix start( )</td><td>Switch Table Accessed Routines

Form:
prefix name(args)
name must be:
<table><tr><th>Character Driver</th><th>Block Driver</th></tr><tr><td>open
close
read
write
ioctl</td><td>open
close
strategy
print</td></tr></table></td><td>Support Routines

Form:
prefix name (args)
name is developer selected
prefix is not needed if the routine is declared static</td><td>proc Routine

Form:
prefix proc (args)
required for TTY drivers performing canonical processing</td></tr>
<tr><th colspan="4">Interrupt Level Routine Type</th></tr>
<tr><td colspan="2">Form:
Block or character driver
prefix int (arg)</td><td colspan="2">Character driver only
prefix rint (arg)
prefix xint (arg)</td></tr>
</table>

close(D2X)

NAME

close — cease access to a device

SYNOPSIS

#include "sys/diskette.h" **[3B2 computer only]**
#include "sys/pump.h"
#include "sys/file.h"
#include "sys/open.h"

*prefix***close**(*dev, flag, otyp*)
dev_t *dev*;
int *flag*;
int *otyp*;

ARGUMENTS

dev device number

flag the flag with which the file was opened. The value does not instruct the driver how to close the file, rather it is a reference to be used as needed. The flag is taken from the **f_flag** member of the `file` structure which is in *file.h*. Refer to the **open**(2) manual page in this section for a listing of the possible flags.

otyp parameter supplied so that the driver can determine how many times a device was opened and for what reasons. The flags assume the **open** routine may be called many times, but the **close** routine should only be called on the last **close** of a device. All flags are defined in *open.h* unless otherwise noted.

- OTYP_BLK - make last close for a block special file
- OTYP_CHAR - make last close for a character special file
- OTYP_MNT - close (unmount) a file system
- OTYP_SWP - close a swapping device
- OTYP_LYR - close a layered process. This flag is used when one driver calls another's **open** or **close** routine. In this case, there is exactly one **close** for each **open** called. This permits software drivers to exist above hardware drivers and removes any ambiguity from the hardware driver regarding how a device is used. This flag applies to block and character devices.

- O_PUMP - close a file that was sent to an intelligent controller on a feature card circuit board (this value is in *pump.h*)
- O_FORMAT - (for the 3B2 computer only) close (unmount) a floppy diskette opened for formatting (this value is in *diskette.h*)

DESCRIPTION

The **close** routine ends the connection between the user process and the previously opened device, and prepares the device (hardware and software) so that it is ready to be opened again.

A device may be opened simultaneously by multiple processes and the **open** driver routine called for each open, but the device will only be closed by the last **close**(2) call (after all forked processes have been closed). The kernel calls the driver **close** routine when the last process using the device issues a **close**(2) call or exits. Block disk devices are closed with an **unmount** system call.

The **close** routine performs the following activities:

- deallocates buffers for private buffering scheme
- unlocks an unsharable device (that was locked in the **open** routine).
- flushes buffers
- notifies device of the close

If an error occurs during **close**, **close** should set the **u.u_error** member of the `user` (D4X) structure. See *open.h* for further information.

IMPORTANT: In a **close** routine, test **u.u_error** before assigning a value. If an error exists already, do not change the value.

A **close** routine should use the flag parameters specified on the **close**(2) manual page when applicable. It should also make the device available for later use by de-allocating resources and cleaning up data structures.

After calling **ttclose**(D3X) for a `tty`(D4X) driver, the driver **close** routine should disconnect the link to the terminal and return to the caller.

SEE ALSO

BCI Driver Development Guide, Chapter 5, "System and Driver Initialization."

init(D2X)

NAME

init — initialize a device

SYNOPSIS

*prefix***init()**

DESCRIPTION

The operating system executes the **init** routine during system initialization. **init** performs the following activities:

- initializes data structures for device access
- allocates buffers for private buffering scheme
- maps device into virtual address space
- initializes hardware (for example, system generation or resetting the board)
- initializes any static data associated with the driver

When the **init** routine is executed, no file systems are accessible. Therefore, **init** should not access disk files, such as getting pump code to send to an intelligent controller.

Use **init** to execute functions when the computer is first brought up; use **start**(D2X) to execute functions after the the computer is brought up and root is mounted.

This routine must be used by local bus boot devices. **init** must never call kernel functions that issue the **sleep(D3X)** function or those that access the `user`(D4X) structure.

SEE ALSO

BCI Driver Development Guide, Chapter 5, "System and Driver Initialization."
start(D2X)

int(D2X)

NAME

int — process a device interrupt

SYNOPSIS

*prefix***int**(*ivec*)
int *ivec*;

ARGUMENT

ivec number or interrupt vector corresponding to the interrupting device. For a 3B2 computer, this number is the physical device number. This number is derived from the bus position of the circuit board (feature card) generating the interrupt. The driver translates the input argument into a logical controller number to access the I/O hardware and locates the status information. For the 3B4000 computer, the argument must be translated to get the logical controller number.

DESCRIPTION

The driver interrupt handler is entered when a hardware interrupt is received from a driver-controlled device. The **int**(D2X) routine is used by both block and character drivers to handle hardware interrupts. The contents of the routine depend on the device for which you are writing your driver.

The **int**(D2X) routine is entered when the CPU receives an interrupt from the device controller. It processes job completions, errors, changes in device status, and spurious interrupts. The **int** routine takes one argument that indicates which interrupt vector generated the interrupt.

The **int** routine for an intelligent controller that does not use individual interrupt vectors for each subdevice must access the completion queue to determine which subdevice generated the interrupt. It must also update the status information, set/clear flags, set/clear error indicators, and so forth to complete the handling of a job. The code should also be able to handle a spurious completion interrupt, identified by an empty completion queue (**logstray(D3X)** is provided for this purpose). When the routine finishes, it should advance the unload pointer to the next entry in the completion queue.

If the driver called **iowait**(D3X) or **sleep**(D3X) to await the completion of an operation, the **int** routine must call **iodone**(D3X) or **wakeup**(D3X) to signal the process to resume.

int is only used with hardware drivers, not software drivers.

CAUTION: The **int** routine must never

- contain calls to the **sleep**(D3X) kernel function
- use functions that call **sleep**
- drop the interrupt priority level below the level at which the interrupt routine was entered
- attempt to access the `user`(D4X) structure
- attempt to access the `proc`(D4X) structure

When an interrupt routine is called, it usually is not being called to interrupt the currently executing user process. So, the interrupt routine must not access the currently executing user process or affect it in any way. For example, the interrupt routine must not access the `user` or `proc` structures of the currently executing user process or call **sleep**.

Table D2X-2 lists kernel functions that must not be called from a driver interrupt routine. These functions access the `user` structure or call **sleep**.

Table D2X−2 Unavailable Interrupt Routine Functions (D3X)

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**	

Establish the goals for your interrupt routine. In general, each interrupt routine contains the following:

- a record of an interrupt occurrence
- an interpretation of the input argument into a meaningful subdevice number and a facility to reject any requests for devices that are not served by the device's controller
- facilities to process interrupts that happened without cause
- proper handling of all possible device errors

In addition, interrupt routines must handle the device-specific tasks that you alone design into the overall plan for your driver. For example,

block driver	dequeue requests, wake up processes sleeping on an I/O request, and ensure that system generation has completed
terminal driver	receive and send characters
printer driver	ensure that characters are sent

The interrupt routine finishes whatever task is left undone before it is called.

SEE ALSO

BCI Driver Development Guide, Chapter 10, "Interrupt Routines."
rint(D2X), **xint**(D2X), **logstray**(D3X)

ioctl(D2X)

NAME

ioctl — control a character device

SYNOPSIS

*prefix***ioctl**(*dev, cmd, arg, mode*)
dev_t *dev*;
int *cmd, arg, mode*;

ARGUMENTS

dev device number

cmd command argument the driver **ioctl** routine interprets as the operation to be performed. The command types vary according to the device.

termio(7) specifies the command types that must work for AT&T terminal drivers. Terminal drivers typically have a command to read the current **ioctl** settings and at least one other that defines new settings. The kernel does not interpret the command type; so, a driver is free to define its own commands.

Create a unique identifying command so your driver can ascertain a correct command has been received. This should be done to guard against misuse by users. For example, AT&T drivers frequently shift a letter eight positions to the left and then OR in a number to create the command. (See TIOC in *termio.h*. Notice in this file how the TCGETA, TCSETA **ioctl** commands are created by ORing a number to TIOC.)

The **ioctl** routine can unshift the command to verify a correct number has been received from the caller. Be sure to comment the command you create.

arg passes parameters between a user program and the driver.

When used with terminals, the argument is the address of a user program structure containing driver or hardware settings. Alternatively, the argument may be an integer that has meaning only to the driver. The interpretation of the argument is driver dependent and usually depends on the command type; the kernel does not interpret the argument.

mode contains values set when the device was opened.

This mode is optional. However, the driver uses it to determine if the device was opened for reading or writing. The driver makes this determination by checking the FREAD or FWRITE setting (values are in *file.h*).

See the *flag* argument description of the **open** routine for further values for the **ioctl** routine's *mode* argument.

DESCRIPTION

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, **ioctl** is used to control device hardware parameters and establish the protocol used by the driver in processing data.

When the user-level program opens a special device file, it can also pass **ioctl** 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. The kernel itself does no processing of the passed command, so it is up to the user program and the driver to agree on what the arguments mean.

I/O control commands are used to implement the terminal settings passed from **getty**(1M) and **stty**(1), to format disk devices, to implement a trace driver for debugging, and to clean up character queues. Since the kernel does not interpret the command type that defines the operation, a driver is free to define its own commands.

Drivers that use an **ioctl** routine typically have a command to "read" the current **ioctl** settings, and at least one other 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 data into the driver itself and 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. The kernel copies it in with the **copyin** function then writes it to the device.

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

1. Define the I/O control command names and the associated value in the driver's header file and comment the commands.
2. Code the **ioctl** routine in the driver that defines the functionality for each I/O control command name that is in the header file.

The **ioctl** routine is coded with instructions on the proper action to take for each command. It is basically a **switch** statement, with each **case** definition corresponding to an **ioctl** name to identify the action that should be taken. However, the command passed to the driver by the user process is an integer value associated with the command name in the header file.

It is critical that command definitions and routines be copiously commented. Because there is so much flexibility in how commands are used, uncommented commands can be very difficult to interpret at a later time.

Terminal drivers use and support the **ioctl** commands defined on the **termio**(7) manual page. For instance, TCGETA gets the parameters associated with the terminal and stores them in the structure referenced in the third argument of the routine call. TCSETA sets the parameters associated with the terminal from the structure referenced in the third argument.

For a complete example of an **ioctl** device driver routine for a networking driver, see the *BCI Driver Development Guide*, Chapter 8, "Input/Output Control (ioctl)."

DEPENDENCIES

Drivers using the **ioctl** routine must have a *c* under the FLAG column in the master file.

SEE ALSO

BCI Driver Development Guide, Chapter 8, "Input/Output Control (ioctl)."

NAME

open — start access to a device

SYNOPSIS

```
#include "sys/file.h"
#include "sys/open.h"
        prefixopen(dev, flag, otyp)
        dev_t dev;
        int flag, otyp;
```

ARGUMENTS

dev device number

flag information passed from the user program **open**(2) or **create**(2) system instructs the driver on how to open the file.

The values for the flag are found in *file.h* associated with the **f_flag** member of the `file` structure. Valid values are:

FAPPEND	open an existing file and set the file pointer to the end of the file
FCREAT	open a new file (ignore if the file already exists)
FEXCL	open a new file, but fail open if the file already exists (used with FCREAT)
FNDELAY	open the file with no delay (do not block the open even if there is a problem)
FREAD	open the file for read-only permission (if ORed with FWRITE, then allow both read and write access)
FSYNC	grant synchronous write permission to a user program for file access
FTRUNC	open an existing file and truncate its length to zero
FWRITE	open a file with write-only permission (if ORed with FREAD, then allow both read and writer access)

otyp parameter supplied so that drivers keep an accurate record of how many times a device is open and for what reasons.

- OTYP_BLK - open a block special file for the first time
- OTYP_CHAR - open a character special file for the first time
- OTYP_MNT - open (mount) a file system
- OTYP_SWP - open a swapping device
- OTYP_LYR - open a layered process. The OTYP_LYR flag is used when one driver calls another's **open** or **close**(D2X) routine. In this case there is exactly one **close** for each **open** called. This permits software drivers to exist above hardware drivers in such a way as to remove any ambiguity from the hardware driver regarding how a device is being used. This flag applies to block and character devices.
- O_PUMP - open a file that is to be sent to an intelligent controller (this value is in *pump.h*)
- O_FORMAT - (for the 3B2 computer only) open (mount) a file to format the diskette (this value is in *diskette.h*)

DESCRIPTION

The **open** routine should perform the following activities:

- validate the minor portion of the device number accessed by the **minor**(D3X) macro
- set up device for subsequent data transfer
- specify whether or not to wait for a hardware connection. Follow the specifications for the **O_NDELAY** flag given on the **open**(2) page of the *Programmer's Reference Manual*. If this flag is set, the **open** will return without waiting for a hardware connection; this is used primarily for software drivers. If it is clear, the **open** will "block" until the hardware establishes a connection.
- verify that if this is an unsharable device, that no other processes are using or sleeping on the device and to then lock the device. An unsharable device is one that should be opened by one process as a time.

The kernel calls the driver **open**(D2X) function as a result of an **open**(2) or **mount**(2) system call for the device file. The **open** routine establishes a connection between the user process issuing the **open** call and the device being opened.

The parameters of the driver **open** routine are the minor device number of the device file and the flags supplied in the *oflag* member of the **open**(2) system call (which map to flag values in the *file.h* header file). The minor device number usually corresponds to the unit number of the physical device being opened.

An **open** routine should use the flag parameter as specified in the **open**(2) manual page when applicable. It should also set the device for subsequent data transfer. When a device is opened simultaneously by multiple processes, the operating system calls the **open** routine for each open.

If an error occurs, the routine sets **u.u_error**. Read and write parameters are defined in *user.h*.

An incorrect special device file could cause the driver **open** routine to be passed an incorrect device number. Through verification, the minor device number is compared to a variable containing the number of devices associated with a controller. This variable is assigned in the driver's initialization routine or in the master file.

Additional **open** routine operation is dependent upon the device being opened. For example, the **open** routine for a removable media disk drive could lock the disk drive door and cause the disk controller to select the drive. Or the **open** routine for a terminal interface controller could turn on data terminal ready (DTR).

SEE ALSO

Section D4X, "Data Structures(D4X)," of this manual.
BCI Driver Development Guide, Chapter 5, "System and Driver Initialization"
minor(D3X), **start**(D2X)

print(D2X)

NAME

print — display a message on a system console

SYNOPSIS

*prefix***print**(*dev, str*)
dev_t *dev*;
char **str*;

ARGUMENTS

dev device number

str character string describing the problem. The nature of the problem contained in *str* should be included in the driver output.

DESCRIPTION

Block drivers must provide a **print** routine to send warning messages from the driver to the console when abnormal situations are detected by the kernel during execution of the **strategy**(D2X) routine. An example of an abnormal situation would be when a disk drive has no more room on the disk.

NOTE: Use the **cmn_err**(D3X) function to send messages to the console. For further information on this function, see Section D3X in this manual.

DEPENDENCIES

Drivers using the **print** routine must have a *b* under the FLAG column in the master file.

SEE ALSO

BCI Driver Development Guide, Chapter 11, "Error Reporting."

proc(D2X)

NAME

proc — process character device-dependent operations

SYNOPSIS

```
prefixproc(tp, cmd)
struct tty *tp;
int cmd;
```

ARGUMENTS

tp pointer to the `tty`(D4X) structure

cmd an operation that the **proc** routine performs. Typically, the driver encodes a **case** statement for each command with code to perform the operations that are described as follows. Refer to the **proc** routine in Appendix D of the *BCI Driver Development Guide* for an example of how most of the commands are coded.

T_BLOCK send command to the terminal controller to prohibit further input because the input queue has reached the high water mark (buffer is full). This **case** should OR (enable) the TBLOCK flag into the **t_state** member of the `tty` structure.

T_BREAK send a break to a `tty` device. Typically, this command is not handled in the **proc** routine; instead, it is handled in the **ioctl** routine. Refer to the **ioctl** routine in Appendix D of the *BCI Driver Development Guide* for an example.

T_DISCONNECT send a command to the terminal controller to request that it disconnect a terminal device (tell it to drop carrier).

T_INPUT prepare a `tty` device to receive input.

T_OUTPUT initiate output to the device if the device is not busy or output has not been suspended.

T_PARM change parameters in the `tty` structure of a particular device. The driver **proc** routine is called to update the device to the new parameters, if the device is intelligent enough to use the `tty` structure information. The shell layers **sxt** device driver **ioctl** routine calls the **proc** routine of the device with T_PARM when the `tty` structure has been changed.

T_RESUME send command to the terminal controller to indicate that terminal output should be resumed because a CTRL-q character has been received. The TTSTOP bit in the **t_state** member of the `tty` structure should be cleared.

Note that if IXANY is set in the **c_iflag** of the `termio` structure, any character can cause the terminal to resume. See **termio**(7) for further information.

T_RFLUSH — send command to terminal controller to flush terminal input queue. If **t_state** is set to TBLOCK, call the T_UNBLOCK section of the **proc** routine.

T_SUSPEND — send a character to the terminal controller to suspend output to the terminal because a CTRL-s character has been received. The driver **proc** routine should set the TTSTOP bit in **t_state** in the `tty` structure.

T_SWTCH — switch between context layers on the **shl**(1) driver. This case is only used in conjunction with the *sxt.c* driver. Typically, this section of code changes control to channel 0 and wakes up any processes sleeping on:

```
&t_link->chans[0]
```

When the SWTCH character (t_cc[VSWTCH]) is input by the terminal device. The line discipline **ttin** routine checks to see if an input character is equal to t_cc[VSWTCH] (normally CTRL-z), and if so, calls **ttyflush** to flush the input and output buffers (if NOFLSH isn't true in **t_lflag**), and then calls the device driver **proc** routine with the command flag T_SWTCH.

T_TIME — notifies the driver that delay timing for a BREAK, carriage return, and so on, has completed.

T_UNBLOCK — allows further input when the input queue has gone below the high water mark. The driver developer resets TTXOFF and TBLOCK in **t_state** when T_UNBLOCK is used.

T_WFLUSH — clears the transmit buffer characters.

DESCRIPTION

The **proc** routine is called by the TTY subsystem to process various device-dependent operations. This routine is required for a character driver that accesses the `tty` or the `linesw` structures.

DEPENDENCIES

This routine is used only by character drivers written in the TTY subsystem.

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`tty`(D4X)

read(D2X)

NAME

read — read data from a character-access device

SYNOPSIS

*prefix***read(***dev***)**
dev_t *dev*;

ARGUMENTS

dev device number

DESCRIPTION

When **read**(2) is executed, the driver initiates and supervises the data transfer from the device to the user data area. For further information on the `user`(D4X) structure, see Section D4X in this manual.

The **read** routine is accessed through the character device switch table, `cdevsw`(D4X). This assumes the driver indicated is a character driver in the master file (even if the driver accesses a block device). **read** is used for raw data I/O.

When the device being accessed is truly a character device (not a block device being accessed through the character device switch table), the transferred data is usually buffered by the driver. For example, using the `tty`(D4X) structure to get a `cblock`(D4X), the **ttread** and **ttwrite**(D3X) functions handle character driver buffering.

Drivers for low-speed character devices, such as terminals and printers, perform semantic processing of data. These drivers typically use the `clist`(D4X) data structure and the TTY subsystem to perform buffering. These transfer data in and out of the user data area.

Drivers for high-speed character devices such as network interface boards generally set up their own buffering schemes.

DEPENDENCIES

Drivers using the **read** routine must have *c* under the FLAG column in the master file.

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."

NAME

rint — service a receive interrupt

SYNOPSIS

*prefix***rint**(*ivec*)
int *ivec*;

ARGUMENT

ivec a number, or interrupt vector, corresponding to the interrupting device.

For the 3B2 computer, this number is the physical device number. The number is derived from the device bus position generating the interrupt. The driver translates the input argument into a logical controller number to access the I/O hardware and to locate the status information associated with the interrupting device.

For the 3B4000 computer, the argument must be translated to get the logical controller number.

DESCRIPTION

A receive interrupt occurs when a device has data to be read. The CPU determines what action to take by using an entry in the interrupt vector table. The interrupt vector table resides in the kernel in main memory. Each interrupt known to the machine has a table entry. Each interrupt vector table entry includes a text address.

Upon receiving an interrupt, the text address value is loaded into the program counter. The CPU then runs the software located at the text address. A driver indicates the number of interrupts needed in the master file.

Drivers using the **rint** routine must also provide **xint** for transmit interrupts. A driver that uses **rint** and **xint** routines should not use an **int** routine.

CAUTION: The **rint** routine must never

- contain calls to the **sleep** kernel function
- use functions that call **sleep**
- lower the interrupt priority level

below the level that the interrupt routine was entered

- attempt to access the user block (`user`(D4X) structure)
- attempt to access the `proc`(D4X) structure

Because an interrupt is not associated with any user process, any previous local variables set up by the driver are not available.

Do not call the functions shown in Table D2X-2 from **rint**.

DEPENDENCIES

Drivers using the **rint** routine must also call **xint** for transmit interrupts. The **rint/xint** pair will be functional only if the value under the #VEC column is double the value under the #DEV column in the master file.

SEE ALSO

BCI Driver Development Guide, Chapter 10, "Interrupt Routines."
xint(D2X), **int**(D2X)

start(D2X)

NAME

start — start access to a device

SYNOPSIS

*prefix***start()**

DESCRIPTION

The **start** routine is called when a computer starts placing a device into a known state. At the time this routine is called, the developer depends on root being mounted[2], and initialization being complete. The Equipped Device Table (EDT) can be read from a **start** routine.

The **start** routine should perform the following activities:

- initialize data structures for device access
- allocate buffers for private buffering scheme
- download pumpcode to controllers
- map device into virtual address space
- initialize hardware (for example, perform a system generation and reset the board)
- initialize the serial device for character drivers
- initialize any static data associated with the driver

The *root* file system is available to **start**; for devices that are not required to bring the system up, such as a tape controller, the **start** routine might include the system generation statements allowing the operating system to access the controller.

If the number of hardware devices is not in the master file, use **start** to ascertain this count. This information is determined from the EDT and is used in the **open**(D2X) routine to verify that the device number argument is correct.

2. No file system access is permitted on adjuncts.

SEE ALSO

BCI Driver Development Guide, Chapter 5, "System and Driver Initialization."
init(D2X)

NAME

strategy — handle device input and output

SYNOPSIS

*prefix***strategy**(*bp*)
struct buf **bp*;

ARGUMENT

bp buffer header data structure pointer.

The pointer is the address of an instance of the buffer header data structure defined in the system header file *buf.h*. All information about the data transfer is contained in the buffer header. It is also used to return status and error information to the kernel and convey the information to the user.

For further information on `buf`(D4X) header data structure, see Section D4X in this manual.

DESCRIPTION

Block drivers must provide a **strategy** routine to handle the data transfer. **strategy** initiates the transfer. It validates the buffer header information, and generates the device operations required to start the data transfer. The **strategy** routine should also perform the following activities:

- get and validate the device number for hardware drivers
- schedule block I/O request
- call subordinate driver routines (if any) that notify controller an I/O operation is waiting
- If the device can only handle one operation at a time, **strategy** must cause the process to sleep until the device is available. For instance, a sequential access device, like a tape drive, should only handle one read/write operation at a time.
- handle any errors occurring during the read/write operations
- log device errors to the Hard Disk Error Log

If an error occurs, set the **b_error** member of the `buf` structure and set (OR operation) B_ERROR into **b_flags**. Perform a character driver read or write based on the value in **b_flags**. The test should be for B_READ or "not" B_READ. B_WRITE cannot be tested since it is not a flag, it is zero.

Most **strategy** routines do not manipulate the `user` area, although it is possible. Avoid calling functions that manipulate the `user`(D4X) structure. These functions (Table D2X-3) are:

Table D2X−3 Unavailable Strategy Routine Functions

copyin	**longjmp**
copyout	**physck**
drv_rfile	**physio**
fubyte	**subyte**
fuword	**suser**
iomove	**suword**
iowait	**useracc**

strategy uses the following fields in the `buf`(D4X) structure, but it should not set them. Note, however, some of these fields can be manipulated with an OR operator.

b_dev
: contains the major and minor number of the device where the I/O is to occur. The minor number is contained in the 8 low order bits, and the major number is contained in the 5 next low order bits. The 3 high order bits should not be used.

b_blkno
: the block number of the device where the I/O is to occur.

b_bcount
: the number of bytes to be transferred by the I/O operation.

b_un.b_addr
: the address of the data in the buffer. The data array is SBUFSIZE bytes long. SBUFSIZE is defined in *param.h*.

b_flags
: indicates the buffer status. If the B_READ bit is set, then the I/O operation is to read from the device. If the bit is not set, the I/O operation is to write. The driver should OR in changes in status.

DEPENDENCIES

Drivers using the **strategy** routine must have *b* under the FLAG column in the master file.

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
physio(D3X), **print**(D2XX), and `buf`(D4X)

write(D2X)

NAME

write — write data to a character-access device

SYNOPSIS

*prefix***write**(*dev*)
dev_t *dev*;

ARGUMENT

dev device number

DESCRIPTION

When **write**(D2X) is executed, the driver initiates and supervises data transfer from the user data area to the device.

The **write** routine is accessed through the character device switch table, `cdevsw`. This assumes you indicated the driver is a character driver in the master file (even if the driver is designed to access a block device). **write** is used for raw data I/O.

Drivers for low-speed character devices typically use the `clist`(D4X) data structure and the TTY subsystem to perform buffering. These structures and functions transfer information to the user data area.

Drivers for high-speed character devices such as network interface boards generally set up their own buffering schemes.

DEPENDENCIES

Drivers using the **write** routine must have *c* under the FLAG column in the master file.

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."

xint(D2X)

NAME

xint — service a transmit interrupt

SYNOPSIS

*prefix***xint**(*ivec*)
int *ivec*;

ARGUMENT

ivec a number, or interrupt vector, corresponding to the interrupting device.

For the 3B2 computer, this number is the physical device number. This number is derived from the device bus position generating the interrupt. The driver translates the input argument into a logical controller number to access the I/O hardware and to locate the status information associated with the interrupting device.

For the 3B4000 computer, the argument must be translated to get the logical controller number.

DESCRIPTION

A transmit interrupt occurs when data is ready to be written to a character device. The CPU determines the action to take by using an entry in the interrupt vector table.

The interrupt vector table resides in the kernel in main memory. Each interrupt known to the machine has a table entry. Each interrupt vector table entry has a text address. Upon receiving an interrupt, the text address value is loaded into the program counter.

The CPU then runs the software located at the text address. A driver indicates the number of interrupts needed in the master file. See the **int**(D2X) manual page in this section for further information on selecting interrupt routines for the 3B4000 computer.

xint is used only with hardware character drivers.

CAUTION: The **xint** routine must never

- contain calls to the **sleep**(D3X) kernel function
- use functions that call **sleep**
- set the interrupt priority level below the level set when

the interrupt routine was entered

- attempt to access the user block (`user`(D4X) structure)
- attempt to access the `proc`(D2X) structure

Since an interrupt is not associated with any user process, any previous local variables set up by the driver are not available.

xint must not call the functions shown in Table D2X-2.

DEPENDENCIES

Drivers using the **xint** routine must also call **rint** for receive interrupts. The **rint/xint** pair will be functional only if the value under the #VEC column is double the value under the #DEV column in the master file.

SEE ALSO

BCI Driver Development Guide, Chapter 10, "Interrupt Routines."
rint(D2X), **int**(D2X)

Section D3X: Kernel Functions(D3X)

Contents

Introduction

Section D3X describes the driver functions serving as library functions for the driver. The functions are presented on separate pages.

Each driver function manual page contains the following headings

- NAME summarizes the function's purpose
- SYNOPSIS describes the function's entry point in the source code
- ARGUMENTS describes any arguments required to invoke the function
- DESCRIPTION describes general information about the function
- RETURN VALUE describes the return values and messages that may result from invoking the function
- LEVEL indicates from which driver level (base or interrupt) the function can be called
- SEE ALSO indicates functions that are related by usage and sources for further information
- SOURCE FILE indicates the directory and file name location of the function. Kernel source file locations are listed by computer type in Table D3X-1.
- EXAMPLE provides an expansion of the information in a usable context

Table D3X−1 Source File Locations

Computer	*Kernel Source Code*
SBC	/usr/src/uts/3b2100vme
3B2	/usr/src/uts/3b2
3B15	/usr/src/uts/3b15 or /usr/src/uts/com
3B4000 MP	/usr/src/uts/3b15 or /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, consult the documentation supplied with your computer.

Function Categories

The functions can be categorized as follows:

Block I/O

- start I/O, **iowait, geteblk**,and **clrbuf**
- finish I/O, **iodone** and **brelse**
- read and write raw data for a block device, **physck** and **physio**

Character I/O

- read data, **getc, getcb**, and **getcf**
- write data, **putc, putcf**, and **putcb**

TTY Subsystem

- clear buffers, **ttyflush**
- delay a process, **tttimeo, ttywait,** and **ttrstrt**
- I/O control, **ttiocom** and **ttioctl**
- open/close terminal, **ttopen, ttinit,** and **ttclose**
- read from a terminal, **canon, ttin,** and **ttread**
- write to a terminal, **ttout, ttwrite,** and **ttxput**

Memory Management

- allocate memory pages **kseg/unkseg, sptalloc/sptfree**
- break up a DMA request, **dma_breakup**
- get starting address and length of the segment descriptor table, **getsrama/getsramb**
- manage a private buffer scheme **malloc, mapinit, mapwant,** and **mfree**

General

- access a pump code file on a 3B4000 computer, **drv_rfile**
- clear a buffer, **bzero**
- compare integers, **max** and **min**
- convert numbers, **btoc**, **ctob**, and **vtop**
- copy data from a driver to a user program, **copyout**, **subyte**, **suword**, and **iomove**
- copy data from a user program to a driver, **copyin**, **fubyte**, **fuword**, and **iomove**
- copy in kernel space, **bcopy**
- display message, **cmn_err**
- delay/activate a process, **delay**, **sleep/wakeup**, and **timeout/untimeout**
- device number access, **major**, **makedev**,and **minor**
- get interrupt vector on a 3B2 computer, **getvec**
- log errors, **hdeeqd**, **hdelog**, and **logstray**
- prevent/allow interrupts, **spl*/splx**
- return control to user program with error code set, **longjmp**
- signal user-level process(es), **psignal** and **signal**
- verify user access, **suser** and **useracc**

In addition to the previous categories, two functions, **nodev** and **nulldev**, are provided for informational purposes, but are not used directly in a driver.

Summary of Kernel Functions

Table D3X-2 lists the kernel functions and their descriptions in alphabetical order.

Table D3X−2 Kernel Function Summary

Routine	Description	Type †
bcopy(*from,to,bcount*)	copies data between locations in the kernel, for example, from one buffer to another	Gais
brelse(*bp*)	returns buffer to the kernel	Bis
btoc(*bytes*)	returns the number of clicks (swappable memory pages) in the specified number of bytes	Gmis
bzero(*addr,bytes*)	clears memory for a number of bytes	Gais
canon(*tp*)	performs canonical processing	C
clrbuf(*bp*)	erases buffer contents	Bs
cmn_err(*level,format,args*)	displays message	Gis
copyin(*userbuf,driverbuf,cn*)	copies data from user space to the driver	Ga
copyout(*driverbuf,userbuf,cn*)	copies data from the driver to user space	Ga
ctob(*clicks*)	returns the number of bytes in the specified number of clicks (swappable memory pages)	Gmis
delay(*ticks*)	delays for *ticks* clock ticks	Gs
dma_breakup(*strat,bp*)	breaks up DMA requests	Gs
drv_rfile(*D_FILE*)	reads a file into a buffer	G

†
a = Written in Assembly Language
B = Block driver usable
C = Character driver usable
G = Generic (usable for block or character)
i = Can be called from an interrupt routine
m = Macro
s = Can be called from the **strategy** routine

Routine	Description	Type
fubyte(*userbuf*)	copies a byte from user to driver	Ga
fuword(*userbuf*)	copies a word from user to driver	Ga
getc(*clp*)	gets first byte from `clist`	Ci
getcb(*clp*)	gets first `cblock` on `clist`	Ci
getcf()	gets a free `cblock`	Ci
geteblk()	gets an empty buffer	Bs
getsrama(*sid*)	gets starting address of SDT	Gms
getsramb(*sid*)	gets length of SDT	Gms
getvec(*baddr*)	gets an interrupt vector for a given virtual board address	G
hdeeqd(*dev,pdsno,edtyp*)	initializes error logging in the hard disk	Gis
hdelog(*eptr*)	logs a hard disk error	Gis
iodone(*bp*)	signals completion of I/O	Bis
iomove(*cp,bytes,rwflag*)	moves *bytes*	G
iowait(*bp*)	suspends execution during block I/O until I/O completion	B
kseg(*pages*)	allocates memory pages to a process	Gs
logmsg(*message*)	log error message	Gis
logstray(*addr*)	logs spurious interrupt error	Gis
longjmp(*env*)	jumps back	Ga
major(*dev*)	returns major number from device number	Gmis
makedev(*majnum,minnum*)	creates a device number	Gmis
malloc(*mp,size*)	allocates space from a map structure	Gis
mapinit(*mp,mapsize*)	initializes map structure	Gmis
mapwant(*vaddr*)	waits for free buffer	Gmis
max(*int1,int2*)	returns the larger integer	Gais
mfree(*mp,size,index*)	returns space to a map structure	Gis
min(*int1,int2*)	returns the smaller integer	Gais
minor(*dev*)	returns minor number from device number	Gmis
nodev()	returns an error upon access	--
nulldev()	performs no operation	--
physck(*nblocks,rwflag*)	verifies block exists	G
physio(*strat,bp,dev,rwflag*)	calls **strategy** routine for direct block I/O	G
psignal(*p,signal*)	sends signal to a process	Gis
putc(*c,clp*)	puts byte on `clist`	Ci
putcb(*cbp,clp*)	links a `cblock` to the `clist`	Ci

Routine	Description	Type
putcf(*cbp*)	puts `cblock` on free list	Ci
signal(*pgrp,signal*)	sends signal to process group	Gis
sleep(*event,priority*)	suspends execution	Gs
spl*()	suspends or allow interrupts	Gais
splx(*oldlevel*)	restore *oldlevel* of interrupts	Gais
sptalloc(*size,mode,base,flag*)	allocates memory pages	Gs
sptfree(*vaddr,size,flag*)	frees allocated memory pages	Gs
subyte(*userbuf,c*)	copies a byte from driver to user	Ga
suser()	verifies superuser permission mode	G
suword(*userbuf,i*)	copies a word from driver to user	Ga
timeout(*ftn,arg,ticks*)	calls function in *ticks* clock ticks	Gis
ttclose(*tp*)	closes a TTY device	C
ttin(*tp, code*)	move character(s) to raw queue	Ci
ttinit(*tp*)	opens a closed TTY device; initializes `tty` structure with default setting on an initial open	Ci
ttiocom(*tp,cmd,arg,mode*)	changes device parameters	C
ttioctl(*tp,cmd,arg,mode*)	sets device parameters	C
ttopen(*tp*)	opens a TTY device	C
ttout(*tp*)	moves a TTY character from user data space to an output queue	Ci
ttread(*tp*)	moves TTY characters from canonical queue to user	C
ttrstrt(*tp*)	restarts TTY output	Ci
tttimeo(*tp*)	time terminal read request	Ci
ttwrite(*tp*)	moves TTY byte from output queue to transmit buffer	C
ttyflush(*tp,rwflag*)	clears a `cblock` and waken processes sleeping on completion of I/O	Ci

Routine	Description	Type
ttywait(*tp*)	suspends TTY processing until I/O completes	C
ttxput(*tp,ucp,node*)	puts data in TTY output buffer	Ci
unkseg(*vaddr*)	frees previously allocated kernel segment	Gs
untimeout(*id*)	cancels **timeout** with matching ID	Gis
useracc(*base,count,access*)	verify user access to data structures	G
vtop(*vaddr,p*)	translates virtual address to physical	Gs
wakeup(*event*)	resumes suspended execution	Gi

bcopy(D3X)

NAME

bcopy — copy data between address locations in the kernel (byte copy)

SYNOPSIS

#include <sys/types.h>

bcopy(*from, to, bcount*)
caddr_t *from, to*;
int *bcount*;

ARGUMENTS

from source address from which the copy is made

to destination address to which copy is made

bcount the number of bytes (characters) moved

DESCRIPTION

This function copies *bcount* bytes from one kernel address to another. Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move algorithm by how the addresses are aligned. If the input and output addresses overlap, the command executes, but the results may not be as expected.

CAUTION: The *from* and *to* addresses must be within the kernel space. No range checking is made. If an address outside of the kernel space is selected, the driver will corrupt the kernel with undefinable side effects.

Note that **bcopy** should never be used to move data in or out of a user buffer, since it has no provision for handling page faults. The user address space can be swapped out at any time, and **bcopy** always assumes that there will be no paging faults. If **bcopy** attempts to access the user buffer when it is swapped out, the system will crash. It is safe to use **bcopy** to move data within kernel space, since kernel space is never swapped out.

RETURN VALUE

Under all conditions, 0 (zero) is returned.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
copyin(D3X), **copyout**(D3X), **fubyte**(D3X), **fuword**(D3X), **iomove**(D3X), **subyte**(D3X), **suword**(D3X)

SOURCE FILE

ml/misc.s

EXAMPLE

In the following example, an I/O request is made for data stored in a RAM disk. If the I/O operation is a read request, the data is copied from the RAM disk to a buffer (line 7). If it is a write request, the data is copied from a buffer to the RAM disk (line 10). The **bcopy** function is used since both the RAM disk and the buffer are part of the kernel address space.

```
1   #define RAMDNBLK   1000          /* Blocks in RAM disk  */
2   #define RAMDBSIZ   512           /* Bytes per block     */
3   char ramdblks[RAMDNBLK][RAMDBSIZ]; /* Blocks forming RAM disk */
4   ...

5   if (bp->b_flags & B_READ)  /* If read request, copy data from */
6                              /* RAM disk data block to system buffer */
7   bcopy(&ramdblks[bp->b_blkno][0], bp->b_un.b_addr, bp->b_bcount);

8   else                       /* else a write copy data from  */
9                              /* system buffer to RAM disk data block */
10  bcopy(bp->b_un.b_addr, &ramdblks[bp->b_blkno][0], bp->b_bcount);
```

Figure D3X–1 bcopy — Data Copy

brelse(D3X)

NAME

brelse — return buffer to the bfreelist

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>

brelse(*bp*)
struct buf **bp*;

ARGUMENT

bp pointer to the buffer header described in *buf.h*. This is the buffer header address being returned to the kernel's buffer pool.

DESCRIPTION

This block interface function returns a buffer to the buffer pool. First, **brelse** wakes up processes sleeping on the the buffer. After the driver function is finished with the buffer, **brelse** returns the buffer header to a list of free buffers and awakens any processes that called **sleep**(D3X) to wait for a free buffer on the `bfreelist`.

RETURN VALUE

Under normal conditions, there is no return value. If B_ERROR has been set due to an error in an earlier I/O transfer, **b_error** is set to 0 (zero).

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
geteblk(D3X), **clrbuf**(D3X), **iowait**(D3X)

SOURCE FILE

os/bio.c

EXAMPLE

In the following example, an I/O request is made, but a buffer has not been allocated. This can take place in a driver **ioctl**(D2X) routine that needs to download pump code to a device controller. A surplus buffer is allocated from the buffer cache (line 3) and cleared of old data (line 4). The new data is copied into the buffer, and the physical I/O is scheduled (device dependent). The driver waits for the completion of the physical I/O operation (line 8). The buffer is then released (line 14).

```
1   register struct buf *bp;
2   ...

3   bp = geteblk;   /* Get an extra (surplus) buffer */
4   clrbuf(bp);     /* Fill buffer with binary zeros */

5   /* Copy data to allocated buffer and        */
6   /* schedule physical I/O request with device */
7   xxstrategy(bp);
8   iowait(bp);                 /* Wait for I/O request to complete */
9   if  ((bp->b_flags &B_ERROR) != 0)  /* If an error occurred with   */
10  {                           /* I/O operation, get any error*/
11  if((u.u_error = bp->b_error) == 0) /* code. Assign to u_error */
12  u.u_error = EIO; /* If error code not present, set default*/
13  } /* endif */
14  brelse(bp);             /* Release buffer when done */
```

Figure D3X–2 brelse — Releases a Buffer

btoc(D3X)

NAME

btoc — convert bytes to clicks

SYNOPSIS

#include <sys/sysmacros.h>

unsigned
btoc(*bytes*)
unsigned *bytes*;

ARGUMENT

bytes quantity of bytes

DESCRIPTION

This macro returns the number of memory pages (clicks) that are needed to contain a specified number of bytes. For example, if the page size is 2048 bytes, then **btoc(3072)** returns 2. **btoc(0)** returns 0.

RETURN VALUE

A non-negative value is always returned.

LEVEL

Base or Interrupt

SEE ALSO

ctob(D3X), **vtop**(D3X)

SOURCE FILE

sysmacros.h

EXAMPLE

Some device controllers accept downloaded microcode. The preferred method of downloading microcode is to allocate a system buffer to hold the microcode. If the microcode is larger then a system buffer, break the code into segments the size of a system buffer and do repetitive moves to download the code. However, some device controllers require all the microcode to be downloaded in a complete unit (segmentation is not permitted). Then the driver can dynamically allocate a private buffer (line 30), but the unit of allocation must be in pages or clicks which is converted (line 30). The microcode is copied to the allocated memory space (line 35). If an invalid address is found, an error condition is returned (line 38). Otherwise, the microcode is downloaded to the controller (line 44) and the private buffer is deallocated (line 46).

```
1   struct  device              /* Physical device registers layout */
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 in  */
7                              /* 0xf0; subdevice reporting in 0x0f  */
8         paddr_t  addr;       /* Data address to be read/written */
9         int      count;      /* Date amount to be read/written  */
10  }; /* end device */

11  struct ucode                /* Microcode input structure layout */
12  {
13        int      count;      /* Number of microcode bytes */
14        char     *code;      /* Microcode location */
15  }; /* end ucode */
16  extern struct device *xx_addr[];  /* Physical device registers */
17  extern int           xx_cnt;      /* Number of devices */
18  ...
19  xx_ioctl(dev, cmd, arg, flag)
20  dev_t  dev;
21  caddr_t arg;
22  {
23  register struct device *rp;
```

Figure D3X–3 btoc — Converts Bytes *(part 1 of 2)*

```
24  switch(cmd)
25  {
26      case XX_DOWNLOAD:
27      {
28          register struct *ucp = (struct ucode *)arg; /* Get microcode */
29          register caddr_t bp;            /* private buffer location  */

30          if  ((bp = kseg(btoc(ucp->count)) == 0)   /* Allocate buffer*/
31          {                   /* If insufficient buffer space memory */
32             u.u_error = ENOMEM;          /* return error condition */
33             return;
34          } /* endif */

35          if  (copyin(ucp->code, bp, ucp->count) == -1) /* Copy microcode */
36          {       /* to allocated buffer area; if invalid address */
37             unkseg(bp);              /* Deallocate buffer area  */
38             u.u_error = EFAULT;       /* Return error condition */
39             return;
40          } /* endif */

41          rp = xx_addr[minor(dev) >> 3];  /* Get device registers    */
42          rp->addr = vtop(bp, u.u_procp); /* Setup the location      */
43          rp->count = up_code.buffer_size; /* and size of microcode  */
44          rp->control = XX_DOWNLD;        /* and download it        */
45          delay(HZ * 5);                  /* Wait for completion    */
46          unkseg(bp);                    /* Deallocate buffer area */
47      } /* endblock */
48      break;
49      ...
```

Figure D3X–3 btoc — Converts Bytes *(part 2 of 2)*

bzero(D3X)

NAME

bzero — clear memory for a given number of bytes

SYNOPSIS

#include <sys/types.h>

bzero(*addr, bytes*)
caddr_t *addr*;
int *bytes*;

ARGUMENTS

addr starting virtual address of memory to be cleared (must be an even-word address)

bytes the number of bytes to clear starting at *addr* (should be a word-size multiple number of bytes)

DESCRIPTION

This function clears a contiguous portion of memory by filling the memory with 0's (zeroes).

RETURN VALUE

Under normal conditions, a 0 (zero) is returned. Otherwise, a -1 is returned.

LEVEL

Base and Interrupt

SEE ALSO

bcopy(D3X), **clrbuf**(D3X)

SOURCE FILE

ml/misc.s

NAME

canon — transfer characters from **t_rawq** to **t_canq**

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>
#include <sys/file.h>
#include <sys/termio.h>

    canon(tp)
    struct tty *tp;
```

ARGUMENTS

tp pointer to the current `tty` structure for the device accessed.

DESCRIPTION

This function moves characters from a terminal's raw input buffer to a processed-character buffer and to handle erase, (BREAK,) (DELETE,) and special character processing (known as canonical processing). A terminal may select to either process input a line at a time or a character at a time. The difference as seen by a user program is that for line at a time processing, a read of a terminal does not return until a whole line of input is accumulated. For character at a time processing, a read returns one character. Canonical processing is performed for line at a time processing only.

The ICANON variable (set in **t_lflag**) is enabled to denote that line at a time and canonical processing be performed, or disabled to denote character at a time processing.

The input buffer (or raw queue **t_rawq** in the `tty` structure) contains delimiters to mark the amount of input to be examined.

During the transfer of data from the raw queue to the canonical queue, if ICANON is set, the following character translations are done:

- Erase character processing
- Kill character processing
- End-of-file character processing
- Escaped characters (characters preceded by a backslash "\")
- XCASE processing (uppercase/lowercase presentation)

Refer to **termio**(7) for further information on these translations.

canon is normally called when the characters in **t_rawq** are ready to be processed. However, you can call **canon** before a delimiter is received in the queue. **canon** will call **sleep** to wait on **t_rawq** (at the TTIPRI **sleep** priority). For this reason, **canon** must never be called from an interrupt routine.

The following flags have special meanings to **canon**:

Flag	**Purpose**	**Header File**
CANBSIZ	Maximum line length for a terminal	*param.h*
CARR_ON	Carrier is present	*tty.h*
FNDELAY	Open file without delay	*file.h*
IASLP	Wakeup process when input is done	*tty.h*
ICANON	Perform canonical processing	*termio.h*
RTO	Timeout in progress for raw device	*tty.h*
TACT	Timeout in progress for the device	*tty.h*
TTIPRI	TTY input priority (28) for **sleep**	*tty.h*
VEOF	Same as **termio**(7) EOF	*termio.h*
VEOL	Same as **termio**(7) NL	*termio.h*
VEOL2	Same as **termio**(7) EOL	*termio.h*
VERASE	Same as **termio**(7) ERASE	*termio.h*
VKILL	Same as **termio**(7) KILL	*termio.h*
VMIN	Same as **termio**(7) MIN	*termio.h*
VTIME	Same as **termio**(7) TIME	*termio.h*
XCASE	Upper/lowercase presentation mode	*termio.h*

Traditionally, **canon** is called by a line discipline **read** routine to transfer characters if there are no characters in the **t_canq**. **canon** is called from the **ttread** line discipline routine to do this.

RETURN VALUE

In general, **canon** calls **sleep** if there is not yet a delimiter in the input **t_rawq**, unless non-canonical processing is in effect. When a delimiter is present, **canon** processes characters until the first delimiter is hit and then returns.

canon returns if

- ICANON is on and characters have been transferred into the **t_canq** up to and including the first delimiter, a delimiter being either a "\n", **t_cc**[VEOF], **t_cc**[VEOL], or **t_cc**[VEOL2].
- If the delimiter count is 0 and **t_state** does not have CARR_ON set.
- If the delimiter count is 0 and the mode of the read has no delay (FNDELAY) set. In this case **u.u_error** is set to EAGAIN and **canon** returns -1.
- If ICANON is not set, and the input parameters **t_cc**[VMIN] (the minimum number of characters to be input) and **t_cc**[VTIME] (the time in tenths of seconds to wait between characters, after the first character has been input) have been satisfied. If **t_cc**[VTIME] is non-zero, and **t_cc**[VMIN] characters have not yet been input, **canon** calls **tttimeo** to schedule a **wakeup** and then calls **sleep.**

  If **canon** must call **sleep** before returning, it passes **sleep** the address of **t_rawq** as the event and sets a priority of TTIPRI (28).

LEVEL

Base Only

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
ttread(D3X), **ttin**(D3X)

SOURCE FILE

io/tty.c

EXAMPLE

The following example uses **canon** from a driver **read** routine.

```
1   /* line discipline read routine for xx terminal */

2   xxread(tp)
3   register struct tty *tp;
4   {
5         register struct clist *tq;

6         tq = &tp->t_canq;

7   /* If no characters to process in the canonical queue, call canon to
8     transfer characters or sleep until a delimiter is present. */

9         if (tq->c_cc == 0)
10              canon(tp);
11        while (u.u_count!=0 && u.u_error==0) {


12        /* transfer characters to user data space from canq */

13        }
14  }
```

Figure D3X−4 canon — Example

clrbuf(D3X)

NAME

clrbuf — erase the contents of a buffer (clear buffer)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/buf.h>

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

ARGUMENT

bp pointer to the `buf`(D4X) structure

DESCRIPTION

The **clrbuf** function clears the buffer and sets the **b_resid** member of the `buf` structure to 0 (zero).

RETURN VALUE

None.

LEVEL

Base and Interrupt

SEE ALSO

geteblk(D3X), **brelse**(D3X), `buf`(D4X)

SOURCE FILE

os/bio.c

EXAMPLE

In the following example, an I/O request is made, but a buffer has not been allocated. This can take place in a driver **ioctl**(D2X) routine that needs to down load pump code to a device controller. A surplus buffer is allocated from the buffer cache (line 3) and cleared of old data (line 4). The new data is copied into the buffer, and the physical I/O is scheduled (device dependent). The driver waits for the completion of the physical I/O operation (line 8). When the I/O operation is finished, an interrupt is generated, and a **wakeup**(D3X) call is made. Any error setting made by the interrupt handler is retrieved, and the buffer is released (line 16).

```
1   register struct buf *bp;
2   ...
3   bp = geteblk;   /* Get an extra (surplus) buffer */
4   clrbuf(bp);     /* Fill buffer with binary zeros */

5   /* Copy data to allocated buffer and        */
6   /* schedule physical I/O request with device */
7   xxstrategy(bp);
8   iowait(bp);     /* Wait for I/O request to complete */
9   if ((bp->b_flags &B_ERROR) != 0)  /* If an error occurred with */
10      {         /* I/O operation get any error */
11          if((u.u_error = bp->b_error) == 0)
12   /* code and assign to u_error */
13              u.u_error = EIO;
14   /* If error code not present set default */
15                                  } /* endif */
16  brelse(bp);  /* Release buffer when finished using it */
```

Figure D3X−5 clrbuf — Fills Buffer with Binary Zeroes

cmn_err(D3X)

NAME

cmn_err — display an error message or trigger a system panic

SYNOPSIS

#include <sys/cmn_err.h>

cmn_err(*level, format, args*)
char **format*;
int *level, args*;

ARGUMENTS

level A constant defined in the *cmn_err.h* header file. *level* indicates the severity of the error condition. The four severity level messages are

CE_CONT	indicates a message should not be preceded with a label such as NOTICE, WARNING, or PANIC. This message form is useful for continuing other messages or for displaying informative messages not connected with an error.
CE_NOTE	reports system events that do not necessarily require user action, but may interest the system administrator. For example, a sector on a disk needing to be accessed repeatedly before it can be accessed correctly might be such an event.
CE_WARN	reports system events requiring immediate attention. If an action is not taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.
CE_PANIC	results in a system panic. Drivers should specify the CE_PANIC level only under the most severe conditions or for debugging a driver. A valid use of CE_PANIC is when the the system cannot continue to function. If the error is recoverable, or not essential to continued system operation, CE_PANIC should not be specified.

format An error message to be displayed. Direct the message to a specific destination by encoding a special character in the first position of the string. Otherwise, the rules for the string are the same as those for **printf** strings. The special characters are as follows:

- ! directs the output of the string only to the **putbuf** (**putbuf** is a circular array in memory accessible with **crash**(1M) used to store messages for analyzing driver performance)
- ^ displays the message only on the console

If a special character is omitted from the first string position, the message is directed to both the **putbuf** and to the console. **cmn_err** appends each *format* with "\n" whether displaying information on the console and/or writing the format message to **putbuf**.

args The set of arguments passed with the message being displayed. Valid conversion specifications are **%s, %u, %d, %o, %x,** and **%D**. **cmn_err** acts similar to **printf**(3S) in displaying messages on the system console or storing in **putbuf**.

NOTE: **cmn_err** does not accept length specifications in conversion specifications. For example, %3**d** is ignored.

DESCRIPTION

At times, a driver may encounter error conditions requiring the attention of a primary or secondary system console monitor. These conditions may mean halting the machine; however, this must be done with caution. Except during the debugging stage, a driver should never stop the system.

The **cmn_err** function with the CE_NOTE argument can be used by driver developers as a driver code debugging tool. However, using **cmn_err** in this capacity can change system timing characteristics.

If CE_PANIC is set, **cmn_err** stops the machine. On the 3B2 computer, the **cmn_err** function automatically handles the use of multiple command windows on the computer console.

RETURN VALUE

No value is returned.

Any message passed to **cmn_err**, unless assigned a specific location, is displayed on the console and assigned to **putbuf**.

If an unknown **level** is passed to **cmn_err**, the following panic error message is displayed:

```
PANIC: unknown level in cmn_err (level=level, msg=format)
```

If there are subsequent panic calls to **cmn_err** after the first panic message is received, DOUBLE PANIC will preface the message(s).

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 11, "Error Reporting."
print(D2X)

SOURCE FILE

os/prf.c

EXAMPLE

The following example shows that the **cmn_err** function can record tracing and debugging information in the **putbuf** (lines 16 and 17); display problems with a device on the system console (line 22); or stop the system if a required device malfunctions (line 28).

```
1   struct  device          /*Physical device registers layout */
2   {
3         int     control; /* Physical device control word  */
4         int     status;  /* Physical device status word   */
5         int     error;     /* Error codes from device    */
6         short    recv_char; /* Receive character from device */
7         short    xmit_char; /* Transmit character to device */
8   }; /* end device */

9   extern struct device xx_addr[]; /* Physical device registers */
10  extern int          xx_cnt;   /* Number of physical devices */
11   ...

12   register struct device *rp;
13   rp = xx_addr[(minor(dev) >> 4) & 0xf)];
14  /* Get device registers */

15  /* Log a message on entry to a function in putbuf */
16   cmn_err(CE_NOTE, "!xx_open function called - dev = 0x%x", dev);
17   cmn_err(CE_CONT, "! flag = 0x%x", flag);
18   /* Continue previous msg */
19  /* Display device power failure on system console */
20   if  ((rp->status & POWER) == OFF)
21   /* If power to device is off */
22            cmn_err(CE_WARN, "^xx_open: Power is OFF on device %d port %d",
23   ((dev >> 4) & 0xf), (dev &0xf));
24  /* endif */

25  /* Halt system when a bad VTOC is found for root device */
26  /* Message is displayed on system console and logged in putbuf */
27                                  if  (rp->error == BADVTOC  &&  dev == rootdev)
28            cmn_err(CE_PANIC, "xx_open: Bad VTOC on root device");

29  /* endif */
```

Figure D3X−6 cmn_err Function

copyin(D3X)

NAME

copyin — copy data from a user program to a driver buffer (copy into kernel)

SYNOPSIS

copyin(*userbuf, driverbuf, cn*)
char **driverbuf, *userbuf*;
int *cn*;

ARGUMENTS

userbuf	user program source address from which data is transferred
driverbuf	driver destination address to which data is transferred (adequate space must be given)
cn	number of bytes transferred

DESCRIPTION

The **copyin** function copies data from a user program to a driver. Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move according to address alignment.

By convention, within the UNIX system kernel, when a driver **read**(D2X) or **write**(D2X) routine is entered, the **u.u_base** member of the `user`(D4X) data structure contains the buffer address in the user address space, and the **u.u_count** member contains the number of bytes remaining to be transferred. After a **read** or **write** call to **copyin** function completes, the driver should increase the value of the **u.u_base** member and decrease the value of the **u.u_count** member by the number of bytes transferred.

RETURN VALUE

Under normal conditions a 0 (zero) is returned indicating the copy is successful. Otherwise, a -1 is returned if one of the following occurs:

- paging fault; the driver tried to access a page of memory for which it did not have read or write access
- invalid user area or stack area
- invalid address that would have resulted in data being copied into the user block

If a -1 is returned, set the **u.u_error** member of the `user`(D4X) structure to EFAULT.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
bcopy(D3X), **copyout**(D3X), **fubyte**(D3X), **fuword**(D3X), **iomove**(D3X), **subyte**(D3X), **suword**(D3X)

SOURCE FILE

ml/misc.s

EXAMPLE

The following example shows that after an appropriate size buffer (line 2) is allocated from a private space management map (line 3), data is copied from the user data area to the private buffer (line 4). If an invalid address is detected in the user data area, the private buffer is released (line 6). and an error code is returned. Otherwise, the pointer to the user data area is advanced to the next starting byte of data to be copied (line 12), and the remaining byte count is updated (line 13).

```
1    while(u.u_count > 0) {  /* While data in user data area,  */
2       cnt = min(u.u_count, MAXBUF);  /* reduce large data output */

3       addr = (caddr_t)malloc(xx_map, cnt); /* Get buffer area from map */

4       if  (copyin(u.u_base, addr, cnt) == -1)  /* Copy data from */
5       {                       /* user to allocated buffer */
6          mfree(xx_map, cnt, addr);  /* If invalid address found, */
7                                  /* release buffer and       */
8          u.u_error = EFAULT;         /* return error code        */
9          return;
10      } /* endif */
11         ...
12      u.u_base  += cnt;     /* Update user data area pointer */
13      u.u_count -= cnt;     /* Update number bytes remaining */
14   } /* endwhile */
```

Figure D3X−7 copyin — Copies Data to Buffer

copyout(D3X)

NAME

copyout — copy data from a driver to a user program (copy out of kernel)

SYNOPSIS

copyout(*driverbuf, userbuf, cn*)
char **driverbuf, *userbuf*;
int *cn*;

ARGUMENTS

driverbuf source address in the driver from which the data is transferred (adequate space must be given)

userbuf destination address in the user program to which the data is transferred (adequate space must be given)

cn number of bytes moved

DESCRIPTION

The **copyout** function copies data from driver buffers to user data space. By convention, within the UNIX system kernel, when a driver **read**(D3X) or **write**(D3X) routine is entered, the **u.u_base** member of the `user`(D4X) data structure contains the address of the buffer in the user address space, and the **u.u_count** member contains the number of bytes remaining to be transferred. After a **read** or **write** call to the **copyout** function completes, the driver should increase the value of the **u.u_base** member and decease the value of the **u.u_count** member by the number of bytes transferred.

Addresses that are word-aligned are moved most efficiently. However, the driver developer is not obligated to ensure alignment. This function automatically finds the most efficient move algorithm according to address alignment.

RETURN VALUE

Under normal conditions a 0 (zero) is returned to indicate a successful copy. Otherwise, a -1 is returned if one of the following occurs:

- memory management fault; the driver tried to access a page of memory for which it did not have read or write access
- invalid user area or stack area
- invalid address that would have resulted in data being copied into the user block, gate tables, user *.text* (addresses where the user does not have write permission)

If a -1 is returned, set the **u.u_error** member of the user structure to EFAULT.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
bcopy(D3X), **copyin**(D3X, **fubyte**(D3X), **fuword**(D3X, **iomove**(D3X), **subyte**(D3X), **suword**(D3X)

SOURCE FILE

ml/misc.s

EXAMPLE

The following example shows that a driver **ioctl**(D2X) routine can be used to get or set device attributes or registers. In the XX_GETREGS condition (line 17), the driver copies the current device register values to a user data area (line 18). If the specified argument contains an invalid address, an error code is returned.

```
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 device xx_addr[]; /* Physical device registers location */
9    ...
10  xx_ioctl(dev, cmd, arg, flag)
11  dev_t   dev;
12  caddr_t arg;
13  {
14  register struct device *rp = &xx_addr[minor(dev) >> 4];
15     switch(cmd)
16     {
17      case XX_GETREGS:  /* Copy device registers to user program */
18           if  (copyout(rp, (struct device *)arg, sizeof(struct device)) == -1)
19               u.u_error = EFAULT;
20           /* endif */
21           break;
22        ...
```

Figure D3X–8 copyout — Specifies User Data Area

ctob (D3X)

NAME

ctob — convert clicks to bytes

SYNOPSIS

#include <sys/sysmacros.h>

unsigned
ctob (*clicks*)
unsigned *clicks*;

ARGUMENT

clicks number of memory pages

DESCRIPTION

This macro returns the number of bytes in the specified number of memory pages (clicks). For example, if the page size is 2048 bytes then **ctob(2)** returns 4096. **ctob(0)** returns 0.

RETURN VALUE

A non-negative value is always returned. However, the number may be truncated if it exceeds the capacity of an unsigned byte.

LEVEL

Base or Interrupt

SEE ALSO

btoc(D3X), **vtop**(D3X)

SOURCE FILE

sysmacros.h

EXAMPLE

In a driver **start**(D3X) routine, a driver can supply its own private buffer area for buffering user data (line 10), but the units of allocation must be given in terms of pages or clicks. If sufficient memory is not available, a message is displayed on the system console, and the system is halted. Otherwise, a space management map is used to manage the allocation and deallocation request of the private buffer area. The space management map must first be initialized with the number of slots that are in the map (line 15). The space management map is used to administer the buffer in terms of bytes (allocated by **kseg**(D3X)). Therefore, compute the number of bytes in the pages (line 17). The allocated private buffer area and its size are assigned to the space management map (line 19).

```
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 D3X−9 ctob — Computes Number of Bytes

delay(D3X)

NAME

delay — delay process execution for a specified number of clock cycles

SYNOPSIS

delay(*ticks*)
int *ticks*;

ARGUMENT

ticks number of clock cycles for a delay. *ticks* are frequently set as an expression containing the system variable HZ (one second) defined in *param.h*.

DESCRIPTION

Occasionally, you may need to wait a given period of time until work is available. The **delay** function provides the wait time. The exact time interval that the delay takes effect cannot be guaranteed, but the value given is a close approximation. The **delay** function calls **timeout**(D3X) schedule and wakeup after the specified amount of time. **wakeup**(D3X) is called after the interval elapses. Upon completion of **timeout, delay** calls **sleep**(D3X) with the same event address that **wakeup** will use after *ticks* of clock ticks. While **delay** is active, **splhi**(D3X) is set. At completion, the former priority level is returned via **splx**(D3X).

RETURN VALUE

None.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 9, "Synchronizing Hardware and Software Events."
iodone(D3X), **iowait**(D3X), **sleep**(D3X), **timeout**(D3X), **ttywait**(D3X), **untimeout**(D3X), **wakeup**(D3X)

SOURCE FILE

os/clock.c

EXAMPLE

Before a driver I/O routine allocates buffers and stores any user data in them, it checks the status of the device (line 11). If the device needs some type of manual intervention (such as, needing to be refilled with paper), a message is displayed on the system console (line 12). The driver waits an allotted time (line 13) for the problem to be corrected before repeating the procedure.

```
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[];  /* physical device registers location */

8    ...
9   register struct device *rp = &xx_addr[minor(dev) >> 4)];
10  /* Get device regs */

11  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 D3X–10 delay — Allows Manual Intervention

dma_breakup(D3X) [3B2, 3B4000 ACP, and SBC Only]

NAME

dma_breakup — break up **physio** request into manageable data chunks

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sbd.h>
#include <sys/immu.h>
#include <sys/fs/s5dir.h>
#include <sys/psw.h>
#include <sys/pcb.h>
#include <sys/region.h>
#include <sys/sysmacros.h>
#include <sys/conf.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/proc.h>
#include <sys/errno.h>
#include <sys/buf.h>
#include <sys/elog.h>
#include <sys/iobuf.h>
#include <sys/systm.h>
#include <sys/inline.h>

        dma_breakup(strat, bp)
        int (*strat)();
        struct buf *bp;
```

ARGUMENTS

strat **strategy**(D2X) routine to call

bp pointer to the `buf` structure

DESCRIPTION

This function breaks up a data transfer request from **physio**(D3X) into chunks of contiguous memory. This function enhances the capabilities of the Direct Memory Access Controller (DMAC). The data is broken into 512 byte sectors until the last data bytes are encountered. **dma_breakup** executes **spl0** around its internal **sleep** calls on reads and writes after the **strategy** routine is called. This may alter previously set **spl*** calls.

The driver must assign the following information before entering this function:

- **b_flags** is set to indicate whether the transfer is a read (B_READ) or a write (B_READ not set).
- **u.u_count** is set to the number of bytes to transfer.
- **u.u_offset** is set to the offset into the file from/to which the data is transferred.
- **u.u_base** is set to the virtual base address for I/O to and from user space.

RETURN VALUE

No value is returned. However, conditions in **dma_breakup** cause the following to be set:

- If temporary buffer memory cannot be allocated, **b_flags** is ORed with B_ERROR and B_DONE, and **b_error** is set to EAGAIN (resource temporarily unavailable). All allocated temporary buffers are deallocated when the transfer completes.
- **u.u_segflg** is altered
- **u.u_base** and **u.u_offset** are incremented by the number of characters transferred, while **u.u_count** is decremented by the number of characters transferred.
- If the I/O transfer is incomplete (**b_flags** does not contain B_DONE), then **b_flags** is set to B_WANTED and **sleep**(D3X) is called to wait until a buffer can be allocated. The **sleep** priority is set to PRIBIO.
- The **sleep** code section is surrounded by a **spl6-spl0** function set which may alter a previously set **spl*** value.
- If B_ERROR is set after the **strategy**(D2X) routine completes, allocated memory is freed and **dma_breakup** returns.
- When the transfer completes, any allocated buffers are freed.

LEVEL

Base Only

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."

SOURCE FILE

io/physdsk.c

EXAMPLE

The following example shows how **dma_breakup** is used from a driver's **read**(D2X) and **write**(D2X) routines.

```
1   struct dsize   {
2           daddr_t nblocks;  /* Number of blocks in disk partition */
3           int     cyloff;   /* Starting cylinder # of partition   */
4   } my_sizes[4] = {

5           20448, 21,        /* partition 0 = cyl 21-305           */
6           21888, 1,         /* partition 1 = cyl  1-305           */
7   };

8   /*    physical read    */

9   my_read(dev)
10  {
11  register int nblks;

12          nblks = my_sizes[minor(dev) & 0x7].nblocks;  /* Get number of blocks */
13                                                      /* blocks in partition  */
14          if (physck(nblks, B_READ)          /* If request is within      */
15          {                                  /* limits for the device,    */
16            physio(my_breakup, 0, dev, B_READ); /* schedule I/O transfer*/
17          }
18  }
```

Figure D3X−11 dma_breakup — Read and Write Access *(part 1 of 2)*

```
19  /*    physical write    */

20  my_write(dev)
21  {
22  register int nblks;

23          nblks = my_sizes[minor(dev) & 0x7].nblocks; /* Get number of blocks */
24                                                     /* blocks in partition  */
25          if (physck(nblks, B_WRITE)              /* If request is within      */
26          {                                    /* limits for the device,    */
27            physio(my_breakup, 0, dev, B_WRITE); /* schedule I/O transfer     */
28          }
29  }

30  /*
31   *    Break up the request that came from physio into chunks of
32   *    contiguous memory.  Pass at least 512 bytes (one sector) at a
33   *    time (except for the last request).
34   */

35  static
36  my_breakup(bp)
37  register struct buf *bp;
38  {
39      dma_breakup(my_strategy, bp);
40  }
```

Figure D3X−11 dma_breakup — Read and Write Access *(part 2 of 2)*

drv_rfile(D3X) [3B4000 and 3B15 computers only]

NAME

drv_rfile — access a file from inside a driver (driver read file)

SYNOPSIS

#include <sys/firmware.h>

drv_rfile (*D_FILE*)
D_FILE **D_FILE*;*

ARGUMENT

D_FILE pointer to a structure that contains

- pointer to the complete path name of the file read
- entry for **drv_rfile** to write the buffer address
- entry for **drv_rfile** to write the buffer size
- entry indicating whether to open or close the file (open = 0 (zero); close = 1)

`D_FILE` is defined in *systm.h*. Table D3X-3 illustrates `D_FILE` structure members.

Table D3X−3 D_FILE Structure Members

Type	Member	Description
char	*file_name	Name of file accessed
char	buffer_address	Buffer address set to zero before open
int	buffer_size	Buffer size set to NULL before open
char	open_close	Open or close flag. open = 0, close = 1

DESCRIPTION

This function reads a file into a buffer that it creates. The buffer address and buffer size are returned. This function should be called twice, once to open and read the file, and again to close the file. When the file is closed, the buffer is released.

This function is useful for bringing a file into a driver, and for accessing files pumped (downloaded) to an intelligent controller. **drv_rfile** can be used with adjuncts through an **ioctl** routine.

CAUTION: Before **drv_rfile** is called, the name of the file to be read must reside in kernel space, not user space.

Before calling **drv_rfile** with the open flag set, set **buffer_address** to NULL. This field must not be altered between open and close requests. Once **drv_rfile** has been given an open request, you must use **drv_rfile** with a close request when you are done, to ensure that the buffer is freed correctly. In addition, the **open_close** flag should not be changed to one between the open and close calls of **drv_rfile**.

RETURN VALUE

The normal completion return value is 0 (zero) which is returned after the file is successfully opened or closed. Otherwise, a -1 is returned when:

- not enough memory is available to read in the file (ENOMEM is placed in **u.u_error**)
- no such file or directory; the indicated file must exist in the root file system (ENOENT is placed in **u.u_error**)
- the file cannot be read from the disk, or the buffer is released (EIO is placed in **u.u_error**)
- the file cannot be copied to internal buffer (EFAULT is placed in **u.u_error**)

Note that the returned **buffer_size** value should not be manipulated as subsequent calls to **drv_rfile** attempt to recreate the buffer and reread the file.

LEVEL

Base Only (Do not call from an interrupt or **init** routine)

SEE ALSO

BCI Driver Development Guide, Chapter 5, "System and Driver Initialization."

SOURCE FILE

os/sys3.c

EXAMPLE

During system initialization, a driver can download microcode to a controller card in a **start**(D2X) (line 4) routine and the `D_FILE` structure is initialized (lines 9 to 12).

```
1   extern struct device *xx_addr[]; /* Physical device registers location*/
2   extern int          xx_cnt;   /* Number of devices */
3   ...
4   xx_start()
5   {
6   register struct device *rp;
7   D_FILE up_code;

8    /* Initialize microprocessor (up) code structure with */
9      up_code.file_name = "/dev/xx_up_code";  /* Name of microcode file */
10     up_code.open_close = 0;       /* Request to read the file &  */
11     up_code.buffer_address = NULL; /* allocate buffer for file   */
12     up_code.buffer_size = 0;

13     if  (drv_rfile(&up_code) == -1)  /* If file read fails, */
14     {          /* display warning on system console   */
15        cmn_err(CE_WARN, "!drv_rfile failed for xx_ device : error code is %d",
16                        u.u_error);
17     } else {
18        for(rp = xx_addr[0]; rp < xx_addr[xx_cnt]; rp++)
19        {             /* For all devices, set up the */
20              rp->addr = vtop(up_code.buffer_address, u.u_procp);
```

Figure D3X−12 A Driver Downloads Microcode *(part 1 of 2)*

```
21              rp->count = up_code.buffer_size; /* location and size of */
22              rp->control = XX_DOWNLOAD;    /* code and download it */
23          } /* endfor */
24      } /* endif */

25      up_code.open_close = 1; /* Set flag to free buffer space */
26      drv_rfile(&up_code);    /* Free buffer space allocated */
27                              /* for microcode file */
28  ...
29  } /* end start */
```

Figure D3X−12 A Driver Downloads Microcode (*part 2 of 2*)

fubyte(D3X) [OBSOLETE]

NAME

fubyte — copy a byte from a user program to a driver (fetch user byte)

SYNOPSIS

```
char
fubyte(userbuf)
char *userbuf;
```

ARGUMENT

userbuf address in a user program area that contains the byte to be moved

DESCRIPTION

This function copies a byte from a user program to a driver.

RETURN VALUE

The normal return value is the requested data byte. Otherwise, a -1 is returned if an attempt is made to overwrite the `user`(D4X) structure or if an attempt is made to overwrite the user stack.

Under normal conditions **fubyte** can return a -1 in the normal data flow. Therefore, if the data accessed by this function may include a -1, use **copyin**(D3X) instead.

If a -1 is returned indicating an error condition, set **u.u_error** to EFAULT.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

bcopy(D3X), **copyin**(D3X), **copyout**(D3X), **fuword**(D3X), **iomove**(D3X), **subyte**(D3X), **suword**(D3X)

SOURCE FILE

ml/misc.s

EXAMPLE

The following example illustrates the use of **fubyte** to move data one byte at a time between the user data area and a `clist`(D4X). As long as there is data in the user data area, you can get the next byte from it (line 7). If the user data area parameter contains an invalid address, an error code is returned. Otherwise, you add the byte to the last `cblock`(D4X) in the `clist` (line 12).

```
1   extern struct tty xx_tty[];
2    ...
3   register struct tty *tp = &xx_tty[minor(dev)];
4   register int    c;

5   while(u.u_count > 0)   /* While there is data in the user data area */
6   {
7      if  ((c = fubyte(u.u_base++)) == -1)  /* Get a byte. */
8      {                          /* If an invalid address  */
9         u.u_error = EFAULT;       /* is found, return error code */
10        return;
11     } /* endif */
12     putc(c, &tp->t_outq);  /* Add byte to output clist */
13     u.u_count--;           /* Update the number of bytes remaining */
14  } /* endwhile */
```

Figure D3X–13 fubyte — Moves Bytes

fuword(D3X) [OBSOLETE]

NAME

fuword — copy a word from a user program to the driver (fetch user word)

SYNOPSIS

```
int
fuword(userbuf)
int *userbuf;
```

ARGUMENT

userbuf user program area address that contains byte to be moved to a driver. This address must be word aligned.

DESCRIPTION

This function copies a single data word from a user program to a driver.

RETURN VALUE

The normal return value is the requested data word. Otherwise, a -1 is returned if an attempt is made to overwrite the `user`(D4X) structure or if an attempt is made to overwrite the user stack (a word alignment error).

Under normal conditions **fuword** can return a -1 in the normal data flow. Therefore, if the accessed data may include a -1, use **copyin**(D3X) instead.

If a -1 (failure) is returned, set **u.u_error** to EFAULT.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

bcopy(D3X), **copyin**(D3X), **copyout**(D3X), **fubyte**(D3X), **iomove**(D3X), **subyte**(D3X), **suword**(D3X)

SOURCE FILE

ml/misc.s

EXAMPLE

When debugging a driver, the **ioctl**(D2X) routine can be used by superusers to manually set a device control register. This can change any incorrect settings made by another driver routine. The new setting is retrieved from the user data area specified by arg (line 24). If arg is an invalid address, an error code is returned. Otherwise, the device control register is assigned the new setting (line 29).

```
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 device xx_addr[]; /* Physical device registers location */
9    ...
10  xx_ioctl(dev, cmd, arg, flag)
11  dev_t               dev;
12  caddr_t arg;
13  {
14  register struct device *rp = &xx_addr[minor(dev) >> 4];
15  register int    c;

16  switch(cmd)
17  {
18   case XX_SETCNTL:
19              if  (u.u_uid && u.u_ruid)  /* Only super user can  */
20              {                          /* set control register  */
```

Figure D3X−14 The fuword Function *(part 1 of 2)*

```
21              u.u_error = EPERM; /* Return permission denied */
22              return;
23            } /* endif */
24            if  ((c = fuword(arg)) == -1) /* Get control setting */
25            {               /* If invalid address is found,*/
26              u.u_error = EFAULT;  /* return error code   */
27              return;
28            } /* endif */
29              rp->control = c;  /* Set device control register */
30            break;
31          ...
```

Figure D3X−14 The fuword Function *(part 2 of 2)*

getc(D3X)

NAME

getc — get a character from a `clist`(D4X)

SYNOPSIS

#include <sys/types.h>
#include <sys/tty.h>

int
getc(*clp*)
struct clist **clp*;

ARGUMENT

clp pointer into the `clist`

DESCRIPTION

The **getc** function receives, as an argument, a pointer to a `clist`. It retrieves the first character from the `clist`, decreases the `clist` character count, and returns the character to the calling routine. If the character taken was the last in the `cblock`(D4X), the `cblock` is returned to the `cfreelist`(D4X). If processes have called **sleep**(D3X) to wait for a free `cblock` from the `cfreelist`, they are awakened after the `cblock` is returned. **getc** manages priority levels thus freeing the driver developer from this concern.

Note you should inhibit interrupts before you manipulate the `tty`(D4X) structure.

RETURN VALUE

The normal return value is the requested character. Otherwise, a -1 is returned when the number of characters in the `clist` is less than one.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
getcb(D3X), **getcf**(D3X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **ttin**(D3X), **ttread**(D3X), `clist`(D4X)

SOURCE FILE

io/clist.c

EXAMPLE

The following example shows that data can be moved between a `clist` and a user data area one byte at a time using **getc**. As long as there is space in the user data area, and there is data in the `clist`, get a single byte from the first `cblock` in the `clist` (line 8) and then copy it to the user data area (line 11).

```
1   extern struct tty xx_tty[];
2   ...
3   register struct tty *tp = &xx_tty[minor(dev)];
4   register int    c;
5   ...
6   while(u.u_count > 0)    /* While there is space in user data area */
7   {
8       if ((c = getc(&tp->t_canq)) == -1)  /* If input queue is empty,*/
9           return;       /* return */
10      /* endif */
11      if (subyte(u.u_base++, c) == -1)  /* Copy character to user  */
12      {                                 /* data area. If invalid   */
13          u.u_error = EFAULT;           /* address is found then   */
14          return;                       /* return error code       */
15      } /* endif */
16      u.u_count--;            /* Update remaining size of data area */
17  } /* endwhile */
```

Figure D3X–15 The getc Function

getcb (D3X)

NAME

getcb — get first `cblock`(D4X) on a `clist`(D4X)

SYNOPSIS

#include <sys/types.h>
#include <sys/tty.h>

```
struct cblock *
getcb(clp)
struct clist *clp;
```

ARGUMENT

clp pointer to a `clist`

DESCRIPTION

The **getcb** function returns the first `cblock` on the `clist` specified by the argument *clp*. **getcb** decreases the `clist` character count by the number of characters in the `cblock` and unlinks the `cblock` from the `clist`.

RETURN VALUE

The normal return value is a pointer to the requested `cblock`. Otherwise, if the `clist` is empty, NULL is returned.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
getc(D3X), **getcf**(D3X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **ttin**(D3X), **ttread**(D3X), `cblock`(D4X)

SOURCE FILE

io/clist.c

EXAMPLE

The following example shows data can be moved in complete `cblocks` between a `clist` and a user data area using **getcb**. As long as there is space in the user data area, and blocks are present in the `clist`, get the first `cblock` in the `clist` (line 9). Next, compute the bytes in the `cblock` and copy the bytes to the user data area (line 15). Finally, the empty `cblock` is returned to the `cfreelist`(D4X) (line 19). If an invalid address is detected, the data transfer returns an error condition.

```
1   extern struct chead cfreelist;
2   extern struct tty xx_tty[];
3    ...
4   register struct tty *tp = &xx_tty[minor(dev)];
5   register struct cblock *cp;
6   register int i;
7   while(u.u_count >= cfreelist.c_size)  /* While user data area *
8   {                   /* has room for an entire cblock, get */
9      if((cp = getcb(&tp->t_canq)) == NULL)  /* an input cblock */
10        return;        /* If clist is empty, then return */
11     /* endif */
12     i = cp->c_last - cp->c_first; /* Get the number of */
13                       /* characters stored in the cblock */
14                       /* Copy data to user          */
15     copyin (u.u_base, (caddr_t)&cp->c_data[cp->c_first], i);
16     u.u_base  += i; /* Increment virtual base addr */
17     u.u_offset  += i;  /* Increment file offset    */
18     u.u_count  -= i;   /* Decrement bytes not transferred */
19     putcf(cp);         /* Return empty cblock to the cfreelist */
20     u.u_base +=i;
21     if  (u.u_error != 0) /* If an invalid address was detected */
22        return;         /* Data transfer returns error */
23     /* endif */
24  } /* endwhile */
```

Figure D3X−16 The getcb Function

getcf(D3X)

NAME

getcf — get a free `cblock`(D4X)

SYNOPSIS

#include <sys/types.h>
#include <sys/tty.h>

struct cblock *
getcf()

DESCRIPTION

The **getcf** function unlinks a `cblock` from the `cfreelist`(D4X) and returns it to the calling routine. **getcf** sets the `cblock` forward pointer to NULL and sets **c_first** to the first character read in the **c_data** array and **c_last** to the last character in the **c_data** array.

RETURN VALUE

Under normal conditions, a pointer to a `cblock` is returned. Otherwise, if the `cfreelist` is empty, **getcf** returns NULL.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
getc(D3X), **getcb**(D3X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **ttin**(D3X), **ttread**(D3X), `cblock`(D4X)

SOURCE FILE

io/clist.c

EXAMPLE

The following example shows that data can be moved in a complete or a partial `cblock` between a user data area and a `clist` with the use of **getcf**. As long as there is data in the user data area, get `cblock` information (line 8). A free `cblock` is obtained from the `cfreelist` (line 10). If the `cfreelist` is empty, set the `cblock` want flag and wait for a free `cblock` (line 13). Then copy the data from the user data area to the allocated `cblock` (line 16).

If an invalid address is detected in the user data area, return the `cblock` to the `cfreelist` (line 18) and return an error code. Otherwise, change the input index **c_last** to the number of the characters in `cblock` and change the output index **c_first** to show that none of the characters have been removed from the `cblock` (line 24). Add the `cblock` to the end of the `clist` (line 26). The pointer to the user data area is advanced to the next starting byte of data to be copied (line 27), and the remaining byte count is updated (line 28). (This example should not be performed in an interrupt routine.)

```
1   extern struct chead cfreelist;
2   extern struct tty xx_tty[];

3   register struct tty *tp = &xx_tty[minor(dev)];
4   register struct cblock *cp;
5   register int    size;

6   while(u.u_count >= 0) /* While data in user data area */
7   {
8       size = min(u.u_count, cfreelist.c_size); /* get smaller buffer */
9       oldlevel = spl4();
10      while((cp = getcf()) == NULL) /* Get free cblock from freelist */
11      {                          /* If freelist empty, then      */
12          cfreelist.c_flag++;    /* set cblock want flag         */
13          sleep(&cfreelist, TTPRIO); /* and wait for free cblock  */
14      /* endwhile */
15      splx(oldlevel);
16      if (copyin(u.u_base, cp->c_data, size) == -1)
17      { /* Copy data from user data area to allocated cblock   */
18        putcf(cp);  /* If an invalid address is detected,     */
19        u.u_error = EFAULT; /* return cblock to cfreelist and */
20        return;     /* return an error code */
21      } /* endif */
22      cp->c_last = size;  /* Record number of bytes in       */
```

Figure D3X–17 The getcf Function *(part 1 of 2)*

```
23                      /* the cblock                   */
24      cp->c_first = 0;    /* Show none of bytes have been   */
25                      /* removed from the cblock      */
26      putcb(cp, tp->t_outq); /* Link cblock to output queue  */
27      u.u_base  += size; /* Update user data area pointer    */
28      u.u_count -= size; /* Update number of bytes remaining */
29  } /* endwhile */
```

Figure D3X−17 The getcf Function *(part 2 of 2)*

geteblk(D3X)

NAME

geteblk — get an empty block

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>

struct buf*
geteblk()

DESCRIPTION

The **geteblk** function retrieves a buffer from the buffer cache and returns the buffer header address to the calling routine. If a buffer header is not available, **geteblk** sleeps until one is available.

When the driver **strategy**(D2X) routine receives a buffer header from the kernel (that is, when the driver is entered through its **strategy**, **read**(D2X), or **write**(D2X) routines), all the necessary members are already initialized. However, when a driver routine allocates buffers for its own use, the routine must set up some of the members before calling the driver **strategy** routine.

The following list explains the state of these members when the buffer header is received from **geteblk** and what must be done.

- **b_flags** is set to B_BUSY to indicate that the buffer is in use. The driver must set the B_READ or B_WRITE flag, depending on the type of transfer. (Leave the B_BUSY flag set and OR the B_READ or B_WRITE flag in place.)
- **b_dev** is set to NODEV and must be initialized by the driver
- **b_bcount** is set to the buffer (SBUFSIZE) byte number
- **b_un.b_addr** is set to the virtual buffer address when the system is started
- **b_blkno** is not initialized by **geteblk**, and must therefore be initialized by the driver

Typically, block drivers do not allocate buffers. The buffer is allocated by the kernel, and the associated buffer header is used as an argument to the driver **strategy** routine. However, in order to implement some driver programs or **ioctl**(D2X) routines, the driver may need its own buffer space. When this is the case, either declare data space in the driver to be used as a buffer; or borrow buffers from the buffer cache.

If the buffer space is not needed frequently, declaring buffer space in the driver (especially for large buffers) is wasteful. Additionally, since block drivers are intimately tied to the buffer cache and the buffer header data structure, using another buffering scheme may require the addition of special case driver code, again expanding the driver unnecessarily. Therefore, in many instances it is advantageous to borrow a buffer from the buffer cache and use the existing driver code to implement special case utilities.

RETURN VALUE

Under normal conditions, a pointer to the `buf`(D4X) structure is returned. Otherwise, the process calls **sleep**(D3X) to wait for a free block. The only possible abnormal return condition occurs if **sleep** fails.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO
BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
`buf`(D4X)

SOURCE FILE

os/bio.c

EXAMPLE

In the following example, an I/O request is made, but a buffer has not been allocated. This can take place in a driver **ioctl** routine that needs to down load pump code to a device controller. A surplus buffer is allocated from the buffer cache (line 3) and cleared of old data (line 4). The new data is copied into the buffer, and the physical I/O is scheduled (device dependent). The driver waits for the completion of the physical I/O operation (line 8). When the I/O operation is finished, an interrupt is generated. Any error setting made by the interrupt handler is retrieved, and the buffer is released (line 14).

```
1   register struct buf *bp;
2   ...

3   bp = geteblk;   /* Get an extra (surplus) buffer */
4   clrbuf(bp);     /* Fill buffer with binary zeros */

5   /* Copy data to allocated buffer and        */
6   /* Schedule physical I/O request with device */
7   xxstrategy(bp);
8   iowait(bp);  /* Wait for I/O request to complete */
9   if  ((bp->b_flags &B_ERROR) != 0)  /* If an error occurred with   */
10  {           /* I/O operation get any error */
11    if((u.u_error = bp->b_error) == 0) /* code; assign to u.u_error */
12      u.u_error = EIO; /* If error code not present set default */
13  } /* endif */
14  brelse(bp);   /* Release buffer when finished using */
```

Figure D3X−18 A Surplus Buffer Is Allocated

getsrama(D3X), getsramb(D3X) [3B4000 Computer Only]

NAME

getsrama, getsramb — get starting address and length of SDT

SYNOPSIS

#include <sys/immu.h>

getsrama(*sid*)
getsramb(*sid*)

ARGUMENT

sid word-aligned segment table virtual address

DESCRIPTION

These two macros facilitate physical-to-virtual address translation on computers equipped with a dual Memory Management Unit (MMU). **getsrama** returns the starting physical address of the Segment Descriptor Table (SDT). **getsramb** returns the length of the SDT. The significance of the function names are that **getsrama** gets register A associated with the SRAM (Segmented Random Access Memory) containing the starting address of the SDT. **getsramb** gets the value of register B containing the physical length of the SDT.

Before using these functions, a section number identification descriptor should be generated. This is done by accessing the **v_sid** member of the VAR (virtual address referencing) structure defined in *immu.h*. Refer to Chapter 4 of the *UNIX Microsystem WE®32100 Microprocessor Information Manual, Maxicomputing in Microspace* for a complete description of virtual address referencing (described in Chapter 1 of this manual).

RETURN VALUE

getsrama returns the SDT starting address; **getsramb** returns the length of the SDT. No error codes are generated.

LEVEL

Base or Interrupt

SOURCE FILE

immu.h

EXAMPLE

In the following example, **maddr** points to the virtual address as provided by the user. Lines 7 through 10 take the address and get the physical address of the segment description table and its length. This information (**psdtpt** and **psdtln**) are passed on to the device. The device uses the information to translate the user's virtual address to a physical address so that user memory can be accessed.

```
1       paddr_t maddr;
2       struct buf *bp;
3
4       maddr = paddr(bp);
5
6       sid = (*(VAR *)&maddr).v_sid;  /* get section ID       */
7       psdtpt = getsrama(sid);                              /* get physical starting  */
8                              /* address of the Segment */
9                              /* Descriptor Table (SDT) */
10      psdtln = getsramb(sid);        /* get length of SDT     */
```

Figure D3X–19 getsrama, getsramb — Example

getvec(D3X) [3B2 computer only]

NAME

getvec — get an interrupt vector for a virtual board address

SYNOPSIS

unsigned char
getvec(*baddr*)
int long *baddr*;

ARGUMENT

baddr a virtual board address

DESCRIPTION

This driver uses this function with the **init**(D3X) routine to get an interrupt vector for a given virtual board address.

RETURN VALUE

Under all conditions, a **unsigned char** numeric value is returned. The only abnormal return value would be a number not logical for the circumstances. If the board address argument is 0 (zero), a divide-by-zero error can occur.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/machdep.c

EXAMPLE

With a 3B2 computer, each device that generates an interrupt must be given an interrupt vector location code. During system initialization, the driver **init** routine gets the interrupt vector location code (line 18) and stores the code in a predefined address on the interface card (an address on the card in the range of 0x0 to 0x200000 can be defined to hold the code).

When a device generates an interrupt, the interface card presents the code to the CPU, which uses it to locate the interrupt handling routine(s) of the driver.

```
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 in */
7                        /* 0xf0; subdevice reporting in 0x0f */
8         paddr_t  addr;    /* Address of data to be read/written */
9         int      count;   /* Amount of data to be read/written  */
10  }; /* end device */

11  extern struct device *xx_addr[]; /* Location of physical */
12                               /* device registers */
13  ...
14  xx_init()
15  {
16  register struct device *rp = xx_addr[0]; /* Get device register */
17                                   /* structure */
18      rp->ivec_num = getvec(xx_addr[0]); /* Get interrupt vector code */

19  } /* end xx_init */
```

Figure D3X–20 Getting An Interrupt Vector

h d e e q d (D 3 X)

NAME

hdeeqd — initialize hard disk error logging

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/dir.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/file.h>
#include <sys/conf.h>
#include <sys/errno.h>
#include <sys/inode.h>
#include <sys/proc.h>
#include <sys/vtoc.h>
#include <sys/hdelog.h>
#include <sys/hdeioctl.h>
#include <sys/cmn_err.h>
#include <sys/open.h>

    hdeeqd(dev, pdsno, edtyp)
    dev_t dev;
    daddr_t pdsno;
    short edtyp;
```

ARGUMENTS

dev device number (contains both the major number and the minor number)

pdsno physical description sector

edtyp error device type. The only valid values are

- EQD_EFC external floppy controller
- EQD_EHDC external hard disk controller
- EQD_ID integral disk drive
- EQD_IF integral floppy disk drive
- EQD_TAPE cartridge tape drive

DESCRIPTION

This function initializes information in the hard disk error logging table for the device specified by *dev*. This function is called once per device.

NOTE: This function is not part of the default set of kernel functions. Ensure that your system has created the current operating system with the *hde.o* object module.

RETURN VALUE

Under all conditions, a 0 (zero) is returned. However, internal errors can occur in **hdeeqd** causing a warning message to display on the console. These errors can be

- The internal major device number is greater than or equal to the number of the controllers (**cdevcnt**) which is assigned by *lboot* when the operating system is loaded. The message is

  `WARNING: hdeeqd: major(ddev) =` *int-major* `(>=cdevcnt)`

  int-major is the internal major device number.

- The count of used disk slots in the logging table exceeds the number of available slots. The message is

  `WARNING: too few HDE equipped slots`
  `bad block handling skipped for maj/min =` *ext-major, ext-minor*

  ext-major and *ext-minor* are the external device numbers.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 11, "Error Reporting."

SOURCE FILE

io/hde.c

EXAMPLE

When a device is being opened for the first time, the driver **open**(D2X) or **init**(D2X) (**open** in this example) routine must identify the device and set up controlling information about the device. In this example, the information is kept on a controlling sector of the disk. If the controlling sector does not exist, the information is encoded as a **static** table in the driver.

```
1   #define XX_CNTLBLKNO  0  /* Controlling sector block number */
2   struct  device          /* Physical device registers layout */
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 */
8         /* number in 0xf0; subdevice reporting in 0x0f  */
9         paddr_t  addr;   /* Data address 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 head pointer */
15        struct buf   *xx_tail; /* I/O buffer queue tail pointer */
```

Figure D3X–21 Hard Disk Error Logging is Initialized *(part 1 of 3)*

```
16          short       xx_flag; /* Logical status flag */
17          struct hdedata xx_edata; /* Disk error log error record */
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 structures 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  ...
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 */
```

Figure D3X−21 Hard Disk Error Logging Is Initialized *(part 2 of 3)*

If this is the first open, the system call made for this device (line 44), initiates error logging for the device (line 47 or 61), allocates a system buffer (line 48), and reads the controlling sector from the **xx_strategy** routine using (line 48). If an error occurred on the read attempt, it displays an error message (**xx_print**) and returns an error condition. Otherwise, saves information from the controlling sector with **bcopy**(D3X) and indicates the device has been opened. Finally, the system buffer is released (line 64).

```
44  if ((dp->xx_flag & XX_OPEN) == 0) /* If first time device opened,*/
45  {
46      register struct buf *bp;
47      hdeeqd(dev, XX_CNTLBLKNO, EQD_ID); /* initialize error logging    */
48      bp = geteblk();          /* Get 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 )
56      {  /* If data error occurred, 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 D3X–21 Hard Disk Error Logging Is Initialized *(part 3 of 3)*

h d e l o g (D 3 X)

NAME

hdelog — log hard disk error

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/dir.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/file.h>
#include <sys/conf.h>
#include <sys/errno.h>
#include <sys/inode.h>
#include <sys/proc.h>
#include <sys/vtoc.h>
#include <sys/hdelog.h>
#include <sys/hdeioctl.h>
#include <sys/cmn_err.h>
#include <sys/open.h>

    hdelog(eptr)
    struct hdedata *eptr;
```

ARGUMENTS

eptr pointer to the `hdedata` structure defined in *hdelog.h*. The driver developer places information in the structure before **hdelog** is called.

DESCRIPTION

This function logs a hard disk error in the error logging queue and displays a warning message on the console to alert the operator to the problem.

The console message is

`WARNING:` *severity readtype* `hard disk error:`
`maj/min =` *external-major-num, external-minor-num*

Where
severity = marginal or unreadable
readtype = CRC (cyclical redundancy check) or ECC (error check and correction)

Call **hdeeqd**(D3X) before this function. **hdelog** logs disk drive media errors.

NOTE: This function is not part of the default set of kernel functions. Ensure that your system has created the current operating system with the *hde.o* object module.

Before calling this function, values must be assigned to the `hdedata` structure. These members include the device number; the disk pack serial number; the physical block address; the type of read operation CRC or ECC; whether the error is marginal or whether the disk is unreadable; the number of unreadable tries; the bit width of the corrected error; and a time stamp.

RETURN VALUE

Under all conditions, a 0 (zero) is returned. However, an internal error can occur in **hdelog** causing a warning message to display on the console. This error occurs when the error logging table is full. In this case, the usual disk error warning message is prefaced with

```
WARNING:  HDE queue full, following report not logged
```

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 11, "Error Reporting."
hdeeqd(D3X), **logstray**(D3X)

SOURCE FILE

io/hde.c

EXAMPLE

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 there has, the driver abandons the I/O request by marking the buffer as being in error, logging an unresolved error (line 58), and marking the I/O operation complete with (line 59). When an error persists in spite of multiple attempts to resolve it, the driver logs marginal errors (line 72) and attempts the I/O operation again. The driver may try to resolve the error with software by using the error correction bits in an ECC (Error Check and Correction) 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 in  */
7                        /* 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        */
17        struct iostat  xx_stat;  /* Unit I/O statistics for       */
18        /* establishing an error rate during error logging       */
19  }; /* end xx_ */
```

Figure D3X–22 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 structures table */
29  extern struct device *xx_addr[]; /* Physical device registers 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  { /* then */
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 as being 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 block number in error*/
53        dp->xx_edata.readtype = HDEECC; /* Error type: error check */
```

Figure D3X–22 hdelog — Logs Media Errors *(part 2 of 3)*

```
54          dp->xx_edata.severity = HDEUNRD; /* Data was 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 */
63          /* leave on I/O queue so that 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 error block number*/
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 D3X−22 hdelog — Logs Media Errors *(part 3 of 3)*

iodone(D3X)

NAME

iodone — resume execution suspended pending block I/O

SYNOPSIS

```
#include <sys/types.h>
#include <sys/buf.h>

    iodone(bp)
    struct buf *bp;
```

ARGUMENT

bp pointer to the block interface buffer structure defined in *buf.h*. This is the address of the buffer header associated with the buffer where the I/O occurred.

DESCRIPTION

iodone is called by the driver interrupt routine when the data transfer is complete. **iodone** does the following:

- awakens the process(es) that called **sleep**(D3X) to wait for the buffer header if I/O is synchronous
- releases the block if I/O is asynchronous and awakens processes awaiting asynchronous I/O
- marks **b_flags** of buffer with B_DONE

RETURN VALUE

Under all conditions, no value is returned.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 9, "Synchronizing Hardware and Software Events."
delay(D3X), **iowait**(D3X), **sleep**(D3X), **timeout**(D3X), **ttywait**(D3X), **untimeout**(D3X), **wakeup**(D3X)

SOURCE FILE

os/bio.c

EXAMPLE

Generally, the first validation test performed by any block device **strategy**(D2X) routine is a check for an end-of-file (EOF) condition. The **strategy** routine is responsible for determining an EOF condition when the device is accessed directly (for example, **physio**(D3X)). If a **read** request is made for one block beyond the limits of the device (line 8), it will report an EOF condition. Otherwise, if the request is outside the limits of the device, the routine will report an error condition. In either case, report the I/O operation as complete (line 20).

```
1   #define RAMDNBLK   1000     /* Number of blocks in RAM disk  */
2   #define RAMDBSIZ   512      /* Number of bytes per block     */
3   char ramdblks[RAMDNBLK][RAMDBSIZ]; /* Blocks that form the RAM disk */

4   ramdstrategy(bp)
5   register struct buf *bp;
6   {
7       register daddr_t blkno = bp->b_blkno; /* Get requested block number */

8       if (blkno < 0 || blkno >= RAMDNBLK) /* If requested block is */
9       { /* outside of the RAM disk limits, test for EOF condition */
10       /* which could result from a direct (physio) request      */

11          if (blkno == RAMDNBLK && bp->b_flags & B_READ)  /* If read */
12          { /* is being made for one block beyond RAM disk limits then  */

13              bp->b_resid -= bp->b_bcount; /* Mark an EOF condition.   */
14      /* The return value for read system call computed by          */
15      /* taking the difference between b_count and b_resid          */

16          } else {    /* Otherwise, an I/O attempt is being         */
17              bp->b_error = ENXIO; /* made beyond limits of the RAM    */
18              bp->b_flags |= B_ERROR;  /* disk; return error condition */
19          } /* endif */
```

Figure D3X–23 iodone — Marks a Completed Operation *(part 1 of 2)*

```
20          iodone(bp); /* Mark I/O operation as being completed (B_DONE) */
21          /* and awaken any processes waiting for this I/O operation    */
22          /* or release buffer for asynchronous (B_ASYNC) request */
23          return;
24      } /* endif */
25  ...
```

Figure D3X–23 iodone — Marks a Completed Operation *(part 2 of 2)*

iomove(D3X) [OBSOLETE]

NAME

iomove — move bytes

SYNOPSIS

iomove(*cp, bytes, rwflag*)
int caddr_t *cp*;
int *bytes, rwflag*;

ARGUMENTS

cp bytes are moved to or from this address.

bytes number of bytes to move. If *bytes* is omitted or set to 0 (zero), no bytes are moved.

rwflag indicates whether a block access is a read or a write. Set to B_WRITE to move bytes from a user to a driver. Set to B_READ to move bytes from a driver to a user.

DESCRIPTION

This function copies bytes from user space to a driver, or from a driver to user space. The **u.u_segflg** (described in *user.h*) determines how the copy is made. This function cannot be called from the driver **init**(D3X) routine.

In addition to moving data, **iomove** adds the number of bytes moved to **u.u_base** and **u.u_offset**. **iomove** also decreases **u.u_count** by the number of bytes moved.

RETURN VALUE

Under all conditions, no value is returned. However, if *rwflag* is B_WRITE and **u.u_segflg** is not equal to 1, and the move fails, then the following occurs

- **u.u_error** is set to EFAULT
- **u.u_base**, **u.u_offset**, and **u.u_count** are not changed

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, Input/Output Operations."
bcopy(D3X), **copyin**(D3X), **copyout**(D3X), **fubyte**(D3X), **fuword**(D3X), **subyte**(D3X), **suword**(D3X)

SOURCE FILE

os/move.c

EXAMPLE

With a RAM disk, direct I/O requests can be handled in the driver's **read**(D2X) and **write**(D2X) routines, as long as the I/O requests are for one or more complete blocks of information.

For either a **read** or **write** request, a test is made to determine if the I/O request is in the limits of the RAM disk (**physck**(D3X)).

For a **read** request, the number of blocks the user data area can contain is computed (line 15). The user data area must be large enough to contain at least one complete block. If it cannot, an error condition will be returned. Otherwise, compute the starting block number (line 19). Copy the requested number of blocks from the RAM disk to the user data area (line 20).

```
1   #define RAMDNBLK   1000    /* Number of blocks in RAM disk  */
2   #define RAMDBSIZ   512          /* Number of bytes per block   */
3   char ramdblks[RAMDNBLK][RAMDBSIZ]; /* Blocks forming the RAM disk */

4   ramdread(dev)          /* Direct read request from a block device */
5   dev_t dev;
6   {
7   register daddr_t blkno; /* Starting block number */
8   register int     nblks; /* Block number to be read with (physio)*/
9           /* I/O from or to a block device, the data must be */
10          /* moved as single complete block or multiples of*/
11          /* complete blocks.                              */
```

Figure D3X–24 The iomove Function *(part 1 of 2)*

```
12    if (physck(RAMDNBLK,B_READ)) /* If the read request is in */
13    {    /* the limits of the RAM disk, copy data to user    */

14      if  ((nblks = u.u_count / RAMDBSIZ)) <= 0)
15      { /* Determine number of blocks to be copied; if user data  */
16        u.u_error = EFAULT; /* area cannot hold complete block from   */
17        return;   /* RAM disk, return error condition*/
18      } /* endif */

19      blkno = u.u_offset / RAMDBSIZ; /* Compute starting block number */
20      iomove(&ramdblks[blkno][0], (nblks * RAMDBSIZ), B_READ);
21      /* Copy data to user  */
22    } /* endif */
23  } /* end ramdread */

24  ramdwrite(dev)          /* Direct write request to a block divide */
25  dev_t dev;
26  {
27  register daddr_t blkno;  /* Starting block number */
28  register int     nblks;  /* Number of blocks written; with direct     */
29                   /* I/O from or to block device, the data must be */
30                  /* moved as single complete block or multiples of  */
31                       /* complete blocks.  */

32    if (physck(RAMDNBLK,B_WRITE)) /* If the write request in the limits */
33    {                        /* of the RAM disk, copy data to user*/

34      if  (u.u_count % RAMDBSIZ != 0)  /* See if there are one or more */
35      {           /* complete blocks to be copied; if not, then     */
36        u.u_error = EFAULT; /* return an error condition.          */
37        return;
38      } /* endif */

39      blkno = u.u_offset / RAMDBSIZ; /* Compute starting block number */
40      iomove(&ramdblks[blkno][0], u.u_count, B_WRITE);/*copy data     */
41    } /* endif */
42  } /* ramdwrite */
```

Figure D3X–24 The iomove Function *(part 2 of 2)*

iowait(D3X)

NAME

iowait — suspend execution pending completion of a block I/O request
(input/output wait)

SYNOPSIS

#include <sys/types.h>
#include <sys/buf.h>
iowait(*bp*)
struct buf **bp*;

ARGUMENT

bp pointer to the block interface buffer structure, *buf.h*, where the awaited data transfer takes place

DESCRIPTION

The kernel provides functions to suspend (**iowait**) and continue (**iodone**(D3X)) execution during block I/O. The **iowait** function is called by driver routines that have allocated their own buffers and are waiting for data transfer to complete.

Until **b_flags** contains B_DONE, **iowait** calls **sleep**(D3X) with the event argument set to a pointer to the `buf`(D4X) structure to wait for I/O completion. **iowait** is awakened by a corresponding call to **iodone** when the transfer completes. Do not call **iowait** from the driver **init**(D2X) or interrupt routine.

RETURN VALUE

Under all conditions, no value is returned. However, this function returns any errors in **u.u_error** that may have occurred while a process is waiting for I/O operations to complete on the `buf` structure. If an error is encountered, but **b_error** equals 0 (zero), **u.u_error** is set to EIO.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 9, "Synchronizing Hardware and Software Events."
delay(D3X), **iodone**(D3X), **sleep**(D3X), **timeout**(D3X), **ttywait**(D3X), **untimeout**(D3X), **wakeup**(D3X)

SOURCE FILE

os/bio.c

EXAMPLE

In the following example, an I/O request is made, but a buffer has not been allocated. This can take place in a driver **ioctl**(D2X) routine that needs to down load pump code to a device controller. A surplus buffer is allocated from the buffer cache (line 3) and cleared of old data (line 4). The new data is copied into the buffer, and the physical I/O is scheduled (device dependent). The driver waits for the completion of the physical I/O operation (line 8). When the I/O operation is finished, an interrupt is generated. Any error setting made by the interrupt handler is retrieved, and the buffer is released (line 14).

```
1   register struct buf *bp;
2   ...

3   bp = geteblk;   /* Get an extra (surplus) buffer */
4   clrbuf(bp);     /* Fill buffer with binary zeros */

5   /* Copy data to allocated buffer and              */
6   /* schedule physical I/O request with device      */
7   xxstrategy(bp);
8   iowait(bp); /* Wait for I/O request to complete  */
9   if  ((bp->b_flags &B_ERROR) != 0) /* If an error occurred with      */
10    {                         /* I/O operation, get any error      */
11      if((u.u_error = bp->b_error) == 0) /* code; assign to u.u_error */
12          u.u_error = EIO; /* If error code not present, set default  */
13    } /* endif */
14  brelse(bp);     /* Release buffer when finished using it */
```

Figure D3X–25 The iowait Function

kseg(D3X)

NAME

kseg — allocate memory pages

SYNOPSIS

char *
kseg(*pages***)**
int *pages*;

ARGUMENT

pages the number of memory pages. *pages* must be in the range of 1 to 64. The number of pages per page table (segment) is 64 (defined by NPGPT in *immu.h*). For the call to succeed, the amount of available resident and swappable memory minus the *pages* must be greater than or equal to the minimum memory value. (The minimum resident memory is in **tune.t_minarmem** and the minimum swappable memory is in **tune.t_minasmem**. Both values are defined in *tuneable.h*.)

DESCRIPTION

This function allocates memory pages to a process. The memory is allocated from the `sptmap` structure described in *map.h*. **kseg** returns the new kernel segment virtual address in bytes. The memory segment is initialized with zeros. The allocated memory is guaranteed to have segment alignment and to always be resident in memory (it is never reclaimed and paged out). This function is generally run from the driver **init**(D2X) or **start**(D2X) routines to allocate memory before starting. **unkseg**(D3X) is used to release the memory allocated with **kseg**.

NOTE: **kseg** uses an entire segment of the kernel map; therefore group your **kseg** requests rather than calling it each time you need a single page of memory.

RETURN VALUE

Under normal conditions, the new memory address is returned. Otherwise, NULL is returned if

- *pages* is not in the range of 1 to 64
- memory cannot be allocated from the `map` by the segment virtual address
- there is insufficient physical memory

The first two instances cause a direct return to the caller. The third instance causes a message to display followed by an immediate return to the caller. The message is

```
NOTICE:  kseg - insufficient memory to allocate pages pages -
                system call failed
```

In addition, if the computer loses track of memory, the computer panics with the message

```
PANIC: kseg - ptmemall failed
```

ptmemall is an internal kernel function that allocates physical memory pages.

If insufficient memory is available, **kseg** will sleep waiting for available memory.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
malloc(D3X), **mapinit**(D3X), **mapwant**(D3X), **mfree**(D3X), **sptalloc**(D3X), **sptfree**(D3X), **unkseg**(D3X)

SOURCE FILE

os/mmgt.c

EXAMPLE

In a driver **start**(D2X) routine, a driver can supply its own private buffer area for buffering user data using **kseg**, but the units of allocation must be given in terms of pages or clicks. If sufficient memory is not available, a message is displayed on the system console, and the system is halted. Otherwise, a space management map manages the allocation and deallocation request of the private buffer area. The space management map must first be initialized with the number of map slots (line 15). The space management map is used to administer the buffer in terms of bytes (allocated by **kseg**). Therefore, compute the number of bytes in the pages (line 17). The allocated private buffer area and its size is assigned to the space management map using (line 19).

```
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 */
4                                   /* for 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 buffer, 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 D3X−26 kseg Allocates Memory Pages

logmsg(D3X) [3B15 Computer and 3B4000 Computer Only]

NAME

logmsg — log an error message

SYNOPSIS

```
#include <sys/param.h>
#include <sys/types.h>
#include <sys/erec.h>
#include <sys/err.h>

    logmsg(message)
    char *message;
```

ARGUMENT

message A message of up to 256 characters enclosed in double quotes

DESCRIPTION

This function is used to place an error message in the */usr/adm/errfile* error file that 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 means of logging errors outside the range of existing error types and when a console may not be available on the computer. This function is frequently used in conjunction with **cmn_err**(D3X) to ensure that an error message is displayed and retained for further analysis. (The number of characters in the string is determined by the EMSGSZ constant defined in *erec.h*.) Messages over 256 characters are truncated.

logmsg automatically appends the message with a NULL character.

The following pieces of information are provided in the error message automatically:

- The error type (E_MSG)
- The time that the error occurred
- The BIC ID if the computer is a 3B4000 adjunct processor

logmsg puts the message in the error slot table (the `errslot` structure defined in *err.h*). The error slot table consists of 8 slots on the 3B4000 MP or 3B4000 EADP. Immediately after the message is placed in the table, the UNIX operating system error handler background program extracts the message from the table and puts it into */usr/adm/errfile*. Even on a fully loaded system, the possibility of running out of error slots is minuscule. The message string is logged into the error slot with the E_MSG error record type (general message to be logged) defined in *erec.h*.

When **logmsg** is called, any processes sleeping on **&err.e.org** are awakened and **err.e_flags** is set to "not" E_SLP.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 11, "Error Reporting."
cmn_err(D3X)

SOURCE FILE

io/errlog.c

EXAMPLE

This example shows a use of **logmsg**.

```
1   /* Log error to /usr/adm/errfile  */

2   logmsg("MYDRIVER:  Invalid ioctl received");
3   cmn_err(CE_NOTE,"MYDRIVER:  Invalid ioctl received");
```

Figure D3X–27 logmsg — Logging an Error

logstray(D3X) [3B15 Computer and 3B4000 Computer Only]

NAME

logstray — log a spurious interrupt

SYNOPSIS

```
#include <sys/param.h>
#include <sys/types.h>
#include <sys/buf.h>
#include <sys/dir.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/utsname.h>
#include <sys/sysmacros.h>
#include <sys/elog.h>
#include <sys/edt.h>
#include <sys/erec.h>
#include <sys/systm.h>
#include <sys/map.h>
#include <sys/err.h>
#include <sys/inline.h>
#include <sys/iobuf.h>
#include <sys/dfdrv.h>

    logstray(addr)
    int addr;
```

ARGUMENT

addr the address of any problem source included in an error record. In an interrupt routine, this argument can be the interrupt vector number.

DESCRIPTION

This function logs errors for which no logical reason can be found. This type of error is referred to as a stray error. This function helps the driver developer define a unusual error type. An error record header is built with the following elements from the `errhdr` structure in *erec.h*:

- the E_STRAY label in the **e_type** member
- the current time in the **e_time** member
- for 3B4000 computer Adjunct processors, the ABUS address is placed in the **e_bicid** member

The error log is in */usr/adm/errfile* and the **errpt**(1M) command is used to print the log.

Any background error logging programs waiting to read */dev/error* are awakened. This function is not part of the default set of kernel functions. Ensure that your system has created the current operating system with the *errlog.o* object module.

RETURN VALUE

No value is returned. If memory cannot be allocated for the error log, **logstray** returns to the caller without creating an error log and without passing any return value.

LEVEL

Interrupt only

SEE ALSO

BCI Driver Development Guide, Chapter 11, "Error Reporting."
hdeeqd(D3X), **hdelog**(D3X)

SOURCE FILE

io/errlog.c

longjmp(D3X)

NAME

longjmp — stop a system call and return the EINTR error message

SYNOPSIS

#include <sys/types.h>

void longjmp(*env*)
label_t *env*;

ARGUMENT

env the stack environment address

DESCRIPTION

This function returns control to a sane location (the address in **u.u_qsav**). **u.u_qsav** is set automatically for the **longjmp** call when the driver is entered from a system entry point routine.

This function is a part of the kernel. It is **not** the same as the **longjmp** system call (part of the **setjmp**(3C) call). Both the code and the number of arguments are different.

longjmp is useful when your code has entered many successive layers of subroutines and you wish to return immediately to an upper layer. If an error occurs during processing in a subroutine, for example, the normal exit method is to return a negative value, and have the calling subroutine detect the error and set another negative return value, and so forth, until the first caller is made aware of the error. **longjmp** provides a quick return to the user program that issued the call to the driver.

When the **sleep**(D3X) process is terminated prematurely by a signal, it is necessary to abort the system call and return to a sane point in the user process. The kernel **longjmp** function provides this capability by transferring control in the kernel back to the user process. The effect seen by the user is the system call returns an error (error code EINTR in **errno**). Thus, when a process that called **sleep** receives a signal, the **sleep** function does not normally return to the routine that called it, but executes **longjmp**.

Drivers calling **sleep** must occasionally perform cleanup operations before **longjmp** is called. Typical items that need cleaning up are locked data structures that should be unlocked when the system call is finished. If the **sleep** priority argument is ORed with the defined constant PCATCH, the **sleep** function does not call **longjmp** on receipt of a signal; instead, it returns the value 1 to the calling routine. If the process that called **sleep** is awakened by an explicit **wakeup**(D3X) call rather than by a signal, the **sleep** call returns 0 (zero).

The **u.u_qsav** area is always set up before a driver is called. Therefore, a driver can always use **longjmp** with **u.u_qsav** to stop normal processing when an error is detected in the base level.

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

ml/cswitch.s

EXAMPLE

Any driver that waits for the completion of an I/O request with a priority greater then PZERO (25) can have the I/O request aborted upon receiving any signal. (PZERO is defined in *param.h.*) However, some drivers, especially in communication networks, need to clear the device of the I/O operation before a stop can take place. This is accomplished by setting the PCATCH bit in the priority field. If the return code value from **sleep** is equal to 1, then the **wakeup** is due to receiving a signal (line 3). The driver can do the necessary cleanup code (replace line 5 with your cleanup code) and stop the I/O request (line 6).

```
1  #define XX_PRIORITY   ((PZERO + 1) | PCATCH) /* Can only catch */
2                    /* signals with a priority > PZERO      */

3      if  (sleep(&event, XX_PRIORITY) == 1) /* If return from sleep */
4      {  /* is due to signal being sent to the process, do       */
5         /* do necessary cleanup driver code                     */

6        longjmp(u.u_qsav);  /* Stop system call I/O request */

7      } /* endif */
```

Figure D3X−28 Stopping a System Call

m a j o r (D 3 X)

NAME

major — return the internal major number from a device number

SYNOPSIS

```
#include <sys/types.h>
#include <sys/sysmacros.h>

    int
    major(dev)
    dev_t dev;
```

ARGUMENT

dev internal device number (contains both the major number and the minor number)

DESCRIPTION

This macro extracts the internal major number from a device number. An internal major number is returned only if your driver is compiled into an object file using the **cc**(1) -DINKERNEL option.

RETURN VALUE

The internal major number.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 3, "Drivers in the UNIX Operating System."
makedev(D3X), **minor**(D3X)

SOURCE FILE

sysmacros.h

EXAMPLE

```
1  dev_t dev;
2  cmn_err(CE_NOTE,"Driver Started.  Internal Major# = %d,
3                   Internal Minor# = %d",major(dev), minor(dev));
```

Figure D3X–29 The major Function

m a k e d e v (D 3 X)

NAME

makedev — make a device number from an external major and external minor device number

SYNOPSIS

```
#include <sys/types.h>
#include <sys/sysmacros.h>

    dev_t
    makedev(majnum, minnum);
    int majnum, minnum;
```

ARGUMENTS

majnum major number

minnum minor number

DESCRIPTION

This macro creates a device number from an external major and external minor device number.

RETURN VALUE

The external device number (contains both the major number and the minor number).

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 3, "Drivers in the UNIX Operating System."
major(D3X), **minor**(D3X)

SOURCE FILE

sysmacros.h

EXAMPLE

```
1   register int i, j;
2   dev_t ddev;
3   ddev = makedev(j, idmkmin(i)); /*idmkmin is a driver routine */
```

Figure D3X–30 The makedev Function

NAME

malloc — allocate space from a private space management map

SYNOPSIS

#include <sys/map.h>

```
uint
malloc(mp, size)
struct map *mp;
int size;
```

ARGUMENTS

mp memory map from where the resource is drawn

size number of units of the resource

DESCRIPTION

Drivers may define private space management maps for allocation of memory space, in terms of arbitrary units, using **malloc**. The system maintains the `map` structure by size and index, computed in units appropriate for the memory map. For example, units may be byte addresses, pages of memory, or blocks. The elements of the memory map are sorted by index, and the system uses the *size* member to combine adjacent objects into one memory map entry. The system allocates objects from the memory map on a first-fit basis. The normal return value is an unsigned integer set to the value of **m_addr** from the `map` structure.

malloc allocates memory from a map; it does not allocate the map itself.

RETURN VALUE

Under normal conditions, **malloc** returns the address of the buffer (as an unsigned integer). Otherwise, **malloc** function returns a 0 (zero) if all memory map entries are already allocated.

LEVEL

Base or Interrupt (only if **mapwant** is not set)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
dma_breakup(D3X), **getsrama**(D3X), **getsramb**(D3X), **kseg**(D3X), **mapinit**(D3X), **mapwant**(D3X), **mfree**(D3X), **sptalloc**(D3X), **sptfree**(D3X), **unkseg**(D3X)

SOURCE FILE

os/malloc.c

EXAMPLE

A driver can supply its own private buffer area for storing user data. When an I/O request is made, the necessary user data buffer space can be allocated from the private buffer area by the means of a space management memory map. If the space allocation cannot be satisfied, then the space want flag is set (line 15). The driver waits for space to be returned to the space management memory map (line 16). The data is copied from the user data area to the allocated buffer (line 21). If an invalid address is detected in the user data area, the allocated buffer is released (line 24), and an error code is returned.

In the following example, the use of the **spl***(D3X) function is inherent to the design of this driver. It is not required by the kernel to protect a private map structure for general use.

```
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();
```

Figure D3X–31 Allocating Space from a Private Map *(part 1 of 2)*

```
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 D3X–31 Allocating Space from a Private Map *(part 2 of 2)*

mapinit(D3X)

NAME

mapinit — initialize a private space management map

SYNOPSIS

#include <sys/map.h>

mapinit(*mp, mapsize*)
struct map **mp*;
int *mapsize*;

ARGUMENTS

mp memory map from where the resource is drawn

mapsize number of entries for the memory map table

DESCRIPTION

The driver must initialize the `map` structure by calling **mapinit**. Two memory map table entries are reserved for internal system use and they are not available for memory map use. The **mapinit** macro does not cause the memory map entries to be labeled available. This must be done through **mfree**(D3X) before objects can actually be allocated from the memory map.

Through the **mapinit** macro drivers may define private space management map for allocation of memory space. The system maintains the memory map list structure by size and index, computed in units appropriate for the memory map. Units may be byte addresses, pages of memory, or blocks. The elements of the memory map are sorted by index. The system uses the size member so that adjacent objects are combined into one memory map entry. The system allocates objects from the memory map on a first-fit basis.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
kseg(D3X), **malloc**(D3X), **mapwant**(D3X), **mfree**(D3X), **sptalloc**(D3X), **sptfree**(D3X), **unkseg**(D3X)

SOURCE FILE

map.h

EXAMPLE

A driver can supply its own private buffer area for buffering user data. A space management memory map can be used to manage the allocation and deallocation request of the private buffer area. The space management must first be initialized with the number of slots that are in the memory map (line 6). The private buffer area that is managed by the space management memory map is assigned to the memory map (line 8).

```
1   #define XX_MAPSIZE   12
2   #define XX_BUFSIZE  2560

3   struct map  xx_map[XX_MAPSIZE]; /* Private buffer for space map  */
4       char xx_buffer[XX_BUFSIZE]; /* Driver xx_ buffer area     */
5    ...
6   mapinit(xx_map, XX_MAPSIZE); /* Initialize space management map  */
7                           /* with number of slots in the map  */
8   mfree(xx_map, XX_BUFSIZE, xx_buffer); /* Initialize map        */
9       /* with total buffer area it is to manage                */
```

Figure D3X–32 Initializing the map Structure

m a p w a n t (D 3 X)

NAME

mapwant — wait for free memory

SYNOPSIS

mapwant(*vaddr*)
unsigned int *vaddr*;

ARGUMENT

vaddr virtual address of the first available buffer

DESCRIPTION

This macro is called when a previous call to **malloc**(D3X) fails and the driver wishes to wait for a free buffer to become available. (After a buffer is freed, the driver developer must call **malloc** again.)

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
kseg(D3X), **malloc**(D3X), **mapinit**(D3X), **mfree**(D3X), **sptalloc**(D3X), **sptfree**(D3X), **unkseg**(D3X)

SOURCE FILE

map.h

EXAMPLE

A driver can supply its own private buffer area for storing user data. When an I/O request is made, the necessary buffer space for the user data can be allocated from the private buffer area by the means of a space management map using **malloc**. If the space allocation cannot be satisfied, then the space want flag is set (line 15), and the driver waits for space to be returned to the space management map (line 16).

```
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];  /* Map for a private buffer */
6         char xx_buffer[XX_BUFSIZE]; /* Driver xx_ private buffer */
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)
14     {  /* Get a buffer, if space is not available, */
15        mapwant(xx_map)++; /* request a wakeup when space is */
16        sleep(xx_map, XX_MAPPRIO);  /* returned. Wait. mfree */
17       /* will check mapwant and supply the wakeup call.    */
18     } /* endwhile */
19     splx(oldlevel);
```

Figure D3X–33 mapwant — Waits for Memory

m a x (D 3 X)

NAME

max — return the larger of two integers

SYNOPSIS

max(*int1, int2*)
int *int1, int2*;

ARGUMENTS

int1, int2 both arguments are integers to be compared

DESCRIPTION

This function returns the larger of two integers.

RETURN VALUE

The larger of the two numbers.

LEVEL

Base or Interrupt

SEE ALSO

min(D3X)

SOURCE FILE

ml/misc.s

EXAMPLE

```
1  extern int tthiwat[]; /* High water marks for cblock allocation base */
2                        /* Baud rate (t_cflag & CBAUD)                 */
3  extern struct tty xx_tty[];
4   ...
5  register struct tty *tp = xx_tty[minor(dev)];
6  register int maxsize;

7      maxsize = max(u.u_count, tthiwat[tp->t_cflag & CBAUD]);
8      /* Get larger allowed buffer size */
```

Figure D3X−34 The max Function

m f r e e (D 3 X)

NAME

mfree — free space back into a private space management map

SYNOPSIS

#include <sys/map.h>

mfree(*mp, size, index*)
struct map **mp*;
int *size, index*;

ARGUMENTS

mp map pointer

size number of units being freed

index index of the first unit of the allocated resource

DESCRIPTION

This function releases space back into a private space management map. It is the opposite of **malloc**(D3X), which allocates space that is controlled by a private map structure.

Drivers may define private space management buffers for allocation of memory space, in terms of arbitrary units, using the **malloc** and **mfree** functions and the **mapinit**(D3X) macro. The drivers must include the file *map.h*. The system maintains the memory map list structure by size and index, computed in units appropriate for the memory map. For example, units may be byte addresses, pages of memory, or blocks. The elements of the memory map are sorted by index, and the system uses the size member so that adjacent objects are combined into one memory map entry. The system allocates objects from the memory map on a first-fit basis. **mfree** frees up unallocated memory for re-use.

RETURN VALUE

Under normal conditions, no value is returned. Otherwise, if the **m_addr** member of the `map` structure is returned as 0 (zero), the following warning message is displayed on the console

`WARNING: mfree map overflow` *mp* `lost` *size* `items at` *index*

Where

- *mp* = the hexadecimal address of the `map` structure
- *size* = the decimal number of buffers freed
- *index* = the decimal address to the first buffer unit freed

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
kseg(D3X), **malloc**(D3X), **mapinit**(D3X), **mapwant**(D3X), **sptalloc**(D3X), **sptfree**(D3X), **unkseg**(D3X)

SOURCE FILE

os/malloc.c

EXAMPLE

A driver can supply its own private buffer area for storing user data. When an I/O request is made, the necessary user data buffer space can be allocated from the private buffer area by the means of a space management memory map. If the space allocation cannot be satisfied, then the space want flag is set (line 15), and the driver waits for space to be returned to the space management memory map (line 16). The data is copied from the user data area to the allocated buffer (line 20). If an invalid address is detected in the user data area, the allocated buffer is released (line 23), and an error code is returned.

```
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 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)
14     { /* Get a buffer, if space is not available, */
15         mapwant(xx_map)++; /* request wakeup when space is  */
16         sleep(xx_map, XX_MAPPRIO); /* returned. Wait. mfree  */
17         /* will check mapwant and supply the wakeup call.    */
18     } /* endwhile */
19     splx(oldlevel);

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

Figure D3X–35 Returning Buffer to Space Management Map

m i n (D 3 X)

NAME

min — return the lesser of two integers

SYNOPSIS

min(*int1*, *int2*)
int *int1*, *int2*;

ARGUMENTS

int1, *int2* both arguments are integers to be compared

DESCRIPTION

This function returns the lesser of two integers.

RETURN VALUE

The lesser of the two numbers.

LEVEL

Base or Interrupt

SEE ALSO

max(D3X)

SOURCE FILE

ml/misc.s

EXAMPLE

The following example illustrates a use of **min**.

```
size = min(u.u_count, cfreelist.c_size); /* Get smaller buffer size */
```

minor(D3X)

NAME

minor — return the internal minor device number from a device number

SYNOPSIS

```
#include <sys/types.h>
#include <sys/sysmacros.h>

    int minor(dev)
    dev_t dev;
```

ARGUMENT

dev device number (contains both the internal major and the internal minor device numbers)

DESCRIPTION

This macro returns the internal minor device number. (An internal minor number is returned only if your driver is compiled into an object file with using the cc(1) -DINKERNEL option.)

RETURN VALUE

The internal minor number.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 3, "Drivers in the UNIX Operating System."
major(D3X), **makedev**(D3X)

SOURCE FILE

sysmacros.h

EXAMPLE

In the following example, the internal minor device number is defined by the driver writer. It contains the number of physical devices controlled by the driver; the physical location of the device; and the possible number of subdevices.

The internal minor number is extracted from the device number (line 14) and is used for the following:

- accesses the device logical structure, such as a `tty` structure
- determines if the physical device slot is equipped
- gets the address of the device registers

```
1   struct  device        /* Physical device registers layout   */
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 device xx_addr[]; /* Physical device registers location */
9   extern int           xx_cnt;    /* Number of physical devices */
10  extern struct tty    xx_tty[];
11   ...
12  register struct tty *tp = xx_tty[minor(dev)]; /* Get device's tty struct */
13  register struct device *rp;

14        if  ((minor(dev) >> 3) > xx_cnt) {/* If device number is out */
15           u.u_error = ENXIO; /* equipped device range, return error*/
16           return;
17        } /* endif */

18        rp = &xx_addr[minor(dev) >> 3]; /* Get device registers */
```

Figure D3X–36 The minor Function

nodev(D3X)

NAME

nodev — indicate a driver routine is missing

SYNOPSIS

```
nodev()
{
    u.u_error = ENODEV;
}
```

DESCRIPTION

This function is an internal function that marks the point(s) in the `cdevsw`(D3X) or `bdevsw`(D3X) where a driver's primary routine was omitted. The description of this function is provided for informational purposes, but the **nodev** function should not be used by the driver developer.

RETURN VALUE

Each time **nodev** is accessed, **u.u_error** is set to ENODEV.

LEVEL

Not called from a driver.

SOURCE FILE

os/subr.c

nulldev(D3X)

NAME

nulldev — perform no operation

SYNOPSIS

```
nulldev()
{
}
```

ARGUMENT

none

DESCRIPTION

This function indicates that a driver routine is not necessary for this particular operation (for example, driver **open**(D2X) routine for */dev/kmem*.

RETURN VALUE

None

LEVEL

Not called from a driver.

SOURCE FILE

os/subr.c

physck(D3X)

NAME

physck — verify the requested block exists

SYNOPSIS

#include <sys/types.h>

physck(*nblocks, rwflag*)
daddr_t *nblocks*;
int *rwflag*;

ARGUMENTS

nblocks number of physical blocks in the partition

rwflag flag indicating whether the access is a read (B_READ) or a write (B_WRITE)

DESCRIPTION

physck is used in the block driver **read**(D2X) and **write**(D2X) routines to verify the user-requested block exists on the requested device.

The driver **read** and **write** routines are called through the `cdevsw` table to perform unbuffered I/O; that is, data is transferred directly between the device and user data space. The kernel provides **physck** to help the driver perform unbuffered I/O operations while maintaining the buffer header as the interface structure. This function is called by both the driver **read** routine and the driver **write** routine. The **physck** and **physio**(D3X) functions perform almost all the work needed to be done by a block driver **read** and **write** routines.

If *nblocks* would take a disk read over the end of a disk partition (**physck** calculates this value from information in the `user` structure), **u.u_count** is decreased by the number of bytes past the end of the partition and the read request is truncated accordingly.

If a write request is calculated by **physck** to extend beyond the *nblocks* boundary, **u.u_error** is set to ENXIO and 0 is returned.

NOTE: **physck** calls **spl0**. This may alter previously set **spl*** calls in your driver.

RETURN VALUE

Under normal conditions, 1 is returned indicating that the block exists. Otherwise, a 0 (zero) is returned and **u.u_error** is set to ENXIO.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
physio(D3X)

SOURCE FILE

os/physio.c

EXAMPLE

In the following example, direct I/O is requested, and a test is made to determine if the I/O request is in the limits of the subdevice or the device partition (line 20). If the request is within the limits, the direct transfer of data to or from the user data area is scheduled with the device **strategy**(D2X) routine through the **physio** function (line 22).

```
1   struct dsize   {
2         daddr_t nblocks; /* Disk partition block number */
3         int     cyloff;  /* Starting cylinder # of partition   */
4   } DISKsizes[16] = {

5         20448,  21,   /* partition 0 = cyl 21-305   */
6         12888,  126,  /*     "     1 = cyl 126-305  */
7         9360,   175,  /*     "     2 = cyl 175-305  */
8         7200,   205,  /*     "     3 = cyl 205-305  */
9         3600,   255,  /*     "     4 = cyl 255-305  */
10        21816,  3,    /*     "     5 = cyl 2-305    */
11        21888,  1,    /*     "     6 = cyl 1-305    */
12        72,     1,    /*     "     7 = cyl 1        */
13  };

14  DISKread(dev)  /* Direct read request from a block device */
15  dev_t dev;
16  {
17  register int nblks;

18    nblks = DISKsizes[minor(dev) & 0x7].nblocks;
19    /* Get number of blocks in the partition */
20    if  (physck(nblks, B_READ))  /* If read request in limits of the   */
21    {                            /* disk partition, schedule a direct I/O */
22        physio(DISKstrategy, 0, dev, B_READ);  /* transfer user area  */
```

Figure D3X−37 physck — Verifies Block Exists *(part 1 of 2)*

```
23      } /* endif */

24  } /* end DISKread */

25  DISKwrite(dev)     /* Direct write request to a block device */
26  dev_t dev;
27  {
28  register int nblks;

29      nblks = DISKsizes[minor(dev) & 0x7].nblocks;
30      /* Get number of blocks in the partition                  */
31      if  (physck(nblks, B_WRITE)) /* If write request in limits   */
32      {                        /* of the disk partition, then    */
33        physio(DISKstrategy, 0, dev, B_WRITE); /* schedule direct */
34                       /* I/O transfer from the user data area  */
35  } /* endif */

36  }/* end DISKwrite */
```

Figure D3X−37 physck — Verifies Block Exists *(part 2 of 2)*

NAME

physio — call **strategy**(D2X) routine to process block interface I/O

SYNOPSIS

#include <sys/types.h>

physio(*strat, bp, dev, rwflag*)
int *(*strat)()*;
struct buf *bp**;
int *dev, rwflag*;

ARGUMENTS

strat address of the driver **strategy** routine

bp address of a buffer header. When called from a driver **read**(D2X) or **write**(D2X) routine, this argument is always 0 (zero). The **physio** function supplies the `buf`(D4X) header.

dev device number. The external device number received as an argument to the driver **read** or **write** routine should be used here. The translation to an internal device number through the **minor**(D3X) macro should be taken care of by the **strategy** routine.

rwflag flag indicating whether the access is a read (B_READ) or a write (B_WRITE). Note that B_WRITE cannot be directly tested as it is 0.

DESCRIPTION

The **physio** function sets up a buffer header describing the user data space. It then locks only the pages you need in memory, calls the driver **strategy** routine, and calls **sleep**(D3X) to wait on the address of the buffer header. When the transfer is complete, **physio** is awakened by the driver interrupt level through **iodone**(D3X). It updates information on the `user`(D4X) data structure and returns to the driver **read** or **write** routine.

The block driver **read** and **write** routines are called through the `cdevsw` table to perform unbuffered I/O; that is, data is transferred directly between the device and user data space. The kernel provides **physio** to help the driver perform unbuffered I/O while maintaining the buffer header as the interface structure. **physio** is called by the driver **read** and **write** routines. With the **physck**(D3X) function, these two functions perform almost all the work to be done by a block driver's **read** and **write** routines.

physio automatically handles memory page locking to ensure that the pages impacted by I/O are not swapped out. In addition, **physio** provides automatic page fault checking during I/O.

NOTE: **physio** calls **spl0**. This may alter previously set **spl*** calls in your driver.

RETURN VALUE

Under normal conditions, no value is returned. Otherwise, **physio** returns any error value that occurred in **u.u_error**. If an error occurred and **b_error** contains 0 (zero), **u.u_error** is set to EIO. In addition, an error in direct memory access (DMA) causes **u.u_error** to be set to EFAULT.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
physck(D3X), **strategy**(D2X)

SOURCE FILE

os/physio.c

EXAMPLE

Refer to the example for **physck**(D3X) for an example of **physio**.

psignal(D3X)

NAME

psignal — send signal to a process

SYNOPSIS

#include <sys/signal.h>

psignal(*p, signal*)
struct proc **p*;
int *signal*;

ARGUMENTS

p pointer to the `proc`(D4X) structure of the process being signaled

signal signal sent; *signal* must be in the range of 1 to (NSIG-1). NSIG and valid signals are listed in *signal.h*.

DESCRIPTION

This function is called by drivers to send a signal to a single process. **psignal** sends a signal to the process whose `proc` structure address is passed as the argument *p*. If the process being sent the signal has called **sleep**(D3X) to wait at a priority higher than PZERO, **psignal** makes the process executable (if PZERO has not been ORed with PCATCH). PZERO is defined in *param.h* and **p_pri** is explained on the **proc**(D4X) structure manual page.

Some drivers need to signal processes of the occurrence of certain events. For example, when a user presses the BREAK key, the driver controlling the device that receives the BREAK must signal all processes associated with the device that the BREAK key has been received.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

signal(D3X)

SOURCE FILE

os/sig.c

EXAMPLE

In the following example, a base level routine detects the telephone carrier to a modem has stopped (line 15). The routine signals this event to the process (line 17).

```
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) &  */
6         /* ring (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[]; /* Physical device register location */
11   ...
12  register struct device *rp = &xx_addr[minor(dev) >> 3]; /* Get device regs */
13  register int    port = minor(dev) & 0x07;   /* Get port number */
14   ...
15     if  ((rp->modem_status & (0x0100 << port)) == 0)
16     {  /* If carrier to the modem has been dropped,   */
17        psignal(u.u_procp, SIGHUP);  /* send hangup signal to process */
18        return;
19     } /* endif */
```

Figure D3X−38 Sending a Hangup Signal to a Process

putc(D3X)

NAME

putc — put character on a `clist`(D4X)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    putc(c, clp)
    char c;
    struct clist *clp;
```

ARGUMENTS

c	character to be placed on a `clist`
clp	pointer to the `clist` data structure

DESCRIPTION

The **putc** function places a character onto the specified `clist`. If a new `cblock`(D4X) is needed because there are none allocated for the `clist` or because the last `clist` is full, **putc** retrieves a new `cblock` from the `cfreelist`(D4X).

RETURN VALUE

Under normal conditions, **putc** links the `cblock` to the `clist`, places the character in the `cblock`, and increases the `clist` character count. Otherwise, if the `cfreelist` is empty, **putc** returns a -1. This tells the calling process it must call **sleep**(D3X) to wait on the `cfreelist`.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`clist`(D4X), **getc**(D3X), **getcb**(D3X), **getcf**(D3X), **putcb**(D3X), **putcf**(D3X)

SOURCE FILE

io/clist.c

EXAMPLE

The following example shows data can be moved one byte at a time between the user data area and a `clist` using **putc**. As long as there is data in the user data area, obtain the next byte (line 7). If the user area contains an invalid address, **fubyte** returns an error code. Otherwise, add the byte to the last `cblock` in the `clist` (line 12).

```
1   extern struct tty xx_tty[];
2    ...
3   register struct tty *tp = &xx_tty[minor(dev)];
4   register int    c;

5   while(u.u_count > 0)    /* While there is data in the user data area, */
6   {
7       if  ((c = fubyte(u.u_base++)) == -1)  /* Get byte from user data */
8       {  /* area.  If an invalid address is found,                  */
9          u.u_error = EFAULT; /* return error code                   */
10         return;
11      } /* endif */
12      putc(c, &tp->t_outq); /* Add byte to output clist              */
13      u.u_count--;          /* Update number of bytes remaining      */
14  } /* endwhile */
```

Figure D3X–39 One Byte Data Move

putcb(D3X)

NAME

putcb — link a `cblock`(D4X) to the `clist`(D4X)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    putcb(cbp, clp)
    struct cblock *cbp;
    struct clist *clp;
```

ARGUMENTS

cbp pointer to `cblock` data structure

clp pointer to `clist` data structure

DESCRIPTION

The **putcb** function is passed as arguments of a pointer to a `cblock` and a pointer to a `clist`. It links the `cblock` to the `clist` and increases the character count in the `clist` head by the number of the characters in the `cblock`.

RETURN VALUE

putcb always returns a 0 (zero).

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`cblock`(D4X), `clist`(D4X), **getc**(D3X), **getcb**(D3X), **getcf**(D3X), **putc**(D3X), **putcf**(D3X)

SOURCE FILE

io/clist.c

EXAMPLE

The following example shows data can be moved in a complete or a partial `cblock` between a user data area and a `clist` using **putcb**. As long as there is data in the user data area, obtain a `cblock` worth of information (line 8).

Get a free `cblock` from the `cfreelist`(D4X) (line 10). If the `cfreelist` is empty, set the `cblock` want flag and wait for a free `cblock` (line 13). Copy the data from the user data area to the allocated `cblock` (line 16). If an invalid address is detected in the user data area, return the `cblock` to the `cfreelist` (line 18) and return an error code. Otherwise, change the input index **c_last** to the number of the characters in `cblock` (line 22). Change the output index **c_first** to show that no characters have been removed from the `cblock` (line 24). Add the `cblock` to the end of the `clist` (line 26). The pointer to the user data area is advanced to the next starting byte of data to be copied (line 27), and the remaining byte count is updated (line 28).

```
1   extern struct chead cfreelist;
2   extern struct tty xx_tty[];

3   register struct tty *tp = &xx_tty[minor(dev)];
4   register struct cblock *cp;
5   register int    size;

6   while(u.u_count >= 0) /* While there is data in the user data area, */
7   {
8       size = min(u.u_count, cfreelist.c_size); /* Get smaller buffer size  */
9       oldlevel = spl4();
10      while((cp = getcf()) == NULL)   /* Get free cblock from freelist */
11      {   /* If freelist is empty, then                          */
12          cfreelist.c_flag++; /* set cblock want flag                */
13          sleep(&cfreelist, TTPRIO); /* and wait for a free cblock    */
14      } /* endwhile */
15      slpx(oldlevel);
```

Figure D3X–40 The putcb Function *(part 1 of 2)*

```
16      if (copyin(u.u_base, cp->c_data, size) == -1) /* Copy data from */
17      {  /* user data area to allocated cblock                    */
18         putcf(cp); /* If an invalid address is detected,          */
19         u.u_error = EFAULT; /* return cblock to cfreelist         */
20         return;          /* and return an error code             */
21      } /* endif */
22      cp->c_last = size; /* Record the number of bytes stored in the */
23                         /* cblock                                   */
24      cp->c_first = 0;   /* Show that none of the bytes have been    */
25                         /* removed from the cblock                  */
26      putcb(cp, tp->t_outq); /* Link cblock to output queue          */
27      u.u_base  += size;     /* Update pointer to user data area     */
28      u.u_count -= size;     /* Update number bytes remaining        */
29  } /* endwhile */
```

Figure D3X–40 The putcb Function *(part 2 of 2)*

putcf(D3X)

NAME

putcf — put `cblock`(D4X) on free list

SYNOPSIS

putcf(*cbp*)
struct cblock **cbp*;

ARGUMENT

cbp pointer to `cblock` data structure

DESCRIPTION

A pointer to a `cblock` is passed to the **putcf** function. The **putcf** function returns the `cblock` to the `cfreelist`(D4X). The function also awakens any processes that called **sleep**(D3X) to wait on the `cfreelist`.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`cblock`(D4X), **getc**(D3X), **getcb**(D3X), **getcf**(D3X), **putc**(D3X), **putcb**(D3X)

SOURCE FILE

io/clist.c

EXAMPLE

The following example shows data can be moved in complete `cblocks` between a `clist`(D4X) and a user data area. As long as there is space in the user data area, and there are blocks present in the `clist`, obtain the first `cblock` in the `clist` (line 9). Compute the bytes in the `cblock` and copy the bytes to the user data area (**iomove**(D3X)). Return the empty `cblock` to the `cfreelist` (line 18). If an invalid address is detected, the data transfer (line 14) returns an error condition.

```
1   extern struct chead cfreelist;
2   extern struct tty xx_tty[];
3    ...
4   register struct tty *tp = &xx_tty[minor(dev)];
5   register struct cblock *cp;
6   register int i;

7   while(u.u_count >= cfreelist.c_size) /* While user data area */
8   {   /* has room for an entire cblock */
9       if((cp = getcb(&tp->t_canq)) == NULL)  /* get an input cblock. */
10          return;   /* If clist is empty, return */
11      /* endif */
12      i = cp->c_last - cp->c_first; /* Get number char stored in cblock */
13      /* Copy data to user */
14      copyin (u.u_base, (caddr_t)&cp->c_data[cp->c_first], i);
15      u.u_base += i;  /*Increment virtual base addr          */
16      u.u_offset += i; /*Increment file offset               */
17      u.u._count -= i; /*Decrement bytes not transferred     */
18      putcf(cp);      /* Return empty cblock to the cfreelist */
19      if  (u.u_error != 0) /* If invalid address detected    */
20          return;       /* during data transfer, return     */
21      /* endif */
22  } /* endwhile */
```

Figure D3X–41 Complete cblock Data Move

NAME

signal — send signal to process group

SYNOPSIS

#include <sys/signal.h>

signal(*pgrp, signal*)
int *pgrp, signal*;

ARGUMENTS

pgrp identification number of the process group being signaled

signal signal to send to the process group; refer to *signal.h* for a list of the appropriate signal values

DESCRIPTION

Some drivers need to signal processes on the occurrence of certain events. For example, when a user presses the BREAK key, the driver controlling the device that receives the character must signal all processes associated with the device that BREAK was received. The kernel provided functions, **signal** and **psignal**(D3X), are used by drivers for this purpose. The **signal** function is called to send signals to all the processes associated with a certain process group. All signals are defined in the system header file *signal.h*.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

psignal(D3X)

SOURCE FILE

os/sig.c

EXAMPLE

In a terminal receive interrupt routine (**rint**(D2X)), data is retrieved from the device receive character register. The data word contains the port that transmitted the character, and is used to locate the corresponding `tty`(D4X) structure.

If the received data word is marked with a framing error (the data is not received correctly), but the character portion is binary 0's (zeros), this signifies a BREAK key was pressed (line 22). Therefore, send an interrupt signal to all processes in the process group (line 24).

```
1   struct  device      /* Physical device register location   */
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[]; /* Logical device structure    */
9   extern struct device xx_addr[]; /* Physical device registers */
10  extern int          xx_cnt;    /* Physical device number     */
11   ...
12  xx_rint(board)
13  int board;      /* The hardware board causing 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 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 structure */
```

Figure D3X−42 Sending Signal to Process Group *(part 1 of 2)*

```
22      if ((c & FRERROR) != 0  &&  (c & 0xff) == 0)  /* If BREAK sent,  */
23      {  /* send an INTERRUPT signal to all processes                  */
24         signal(tp->t_pgrp, SIGINT); /* in terminal group and throw    */
25         ttyflush(tp, (FREAD | FWRITE)); /* away input and output data.*/
26         continue;
27      } /* endif */
28  } /* endwhile */
29   ...
```

Figure D3X−42 Sending Signal to Process Group *(part 2 of 2)*

NAME

sleep — suspend process activity pending execution of an event

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>

    sleep(event, priority)
    caddr_t event;
    int priority;
```

ARGUMENTS

event address (signifying an event) for which the process will wait to be updated

priority priority value that is assigned to the process when it is awakened. If *priority* is ORed with the defined constant PCATCH, the **sleep** function does not call **longjmp**(D3X) on receipt of a signal. Instead, it returns the value 1 to the calling routine.

DESCRIPTION

This function suspends execution of a process to await certain events such as reaching a known system state in hardware or software. For instance, when a process wants to read a device and no data is available, the driver may need to call **sleep** to wait for data to become available before returning to the kernel. This causes the kernel to suspend executing the process that called **sleep** and schedule another process. The process that called **sleep** can be restarted by a call to the **wakeup**(D3X) function with the same *event* specified as that used to call **sleep**.

A driver(with data stored in local variables) may call **sleep** while waiting for an event to occur. Make sure another process will not interrupt the driver and overwrite the local variables.

The *event* address used when calling **sleep** should be the address of a kernel data structure or one of the driver's own data structures. The **sleep** address is an arbitrary address that has no meaning except to the corresponding **wakeup** function call. This does not mean that any arbitrary kernel address should be used for **sleep**. Doing this could conflict with other, unrelated **sleep/wakeup** operations in the kernel. A kernel address used for **sleep** should be the address of a kernel data structure directly associated with the driver I/O operation (for example, a buffer assigned to the driver).

A driver should never use the address of the `user`(D4X) structure for **sleep**.

Before a process calls **sleep**, the driver usually sets a flag in a driver data structure indicating the reason why **sleep** is being called.

The *priority* argument, called the **sleep** priority, is used for scheduling purposes when the process awakens. This parameter has critical effects on how the process that called **sleep** reacts to signals. The **sleep** priorities range from 0 to 39, where higher numerical values indicate lower priority levels. If the numerical value of the **sleep** priority is less than or equal to the constant PZERO (generally set to 25 and defined in the *param.h* header file), then the sleeping processes will not be awakened by a signal. However, if the numerical value is greater than PZERO (values 26 to 39), the system awakens the process that called **sleep** prematurely (that is, before the event on which **sleep** was called occurred) on receipt of a non-ignored signal. it returns the value 1 to the calling routine.

To pick the correct **sleep** priority, base your decision on whether or not the process should be awakened on the receipt of a signal. If the driver calls **sleep** for an event that is certain to happen, the driver should use a priority numerically less than PZERO. (However, you should only use priorities less than or equal to PZERO if your driver is crucial to system operation.)

If the driver calls **sleep** while it awaits an event that may not happen, use a priority numerically greater than PZERO. An example of an event that may not happen is the arrival of data from a remote device. When the system tries to read data from a terminal, the terminal driver might call **sleep** to suspend the current process while waiting for data to arrive from the terminal. If data never arrives, the **sleep** call will never be answered. When a user at the terminal presses the BREAK key or hangs up, the terminal driver interrupt handler sends a signal to the reading process, which is still executing **sleep**. The signal causes the reading process to finish the system call without having read any data. If **sleep** is called with a priority value that is not awakened by signals, the process can be awakened only by a specific **wakeup** call. If that **wakeup** call never happened (the user hung up the terminal), then the process executes **sleep** until the system is rebooted.

Drivers calling **sleep** must occasionally perform cleanup operations before **longjmp** is called. Typical items that need cleaning up are locked data structures that should be unlocked when the system call completes. This is done by ORing *priority* with PCATCH and executing **sleep**. If **sleep** returns a 1, then you can cleanup any locked structures and then call **longjmp** to return to the address in **u.u_qsav** (and automatically set **u.u_error** to EINTR).

CAUTION: If **sleep** is called from the driver **strategy**(D2X) routine, you should OR the *priority* argument with PCATCH or select a *priority* of PZERO or less. Should neither be used, catastrophic results could occur if **sleep** ever needed to call **longjmp** (since the state of the `user` structure is not stable in the **strategy** routine).

RETURN VALUE

If the **sleep** *priority* argument is ORed with the defined constant PCATCH, the **sleep** function does not call **longjmp** on receipt of a signal; instead, it returns the value 1 to the calling routine. If the process put in a wait state by **sleep** is awakened by an explicit **wakeup** call rather than by a signal, the **sleep** call returns 0 (zero).

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 9, "Synchronizing Hardware and Software Events."
delay(D3X), **iodone**(D3X), **iowait**(D3X), **timeout**(D3X), **ttywait**(D3X), **untimeout**(D3X), **wakeup**(D3X)

SOURCE FILE

os/slp.c

EXAMPLE

Sometimes a driver must suspend the execution of the current process while it waits for the availability of a hardware or software resource. When a request is made for a buffer area in a private space management memory map (line 9), and the space allocation cannot be satisfied, the space want flag is set (line 11). The driver waits for space to be returned to the space management memory map (line 12).

```
1   struct map  xx_map[XX_MAPSIZE]; /* Private buffer map              */
2         char xx_buffer[XX_BUFSIZE]; /* Driver xx_ private buffer area */
3    ...
4   register caddr_t addr;
5   register int     size;
6      size = min(u.u_count, XX_MAXSIZE); /* Break large I/O request into */
7                                  /* small ones                  */
8      oldlevel = spl4();
9      while((addr = (caddr_t)malloc(xx_map, size)) == NULL) /* Get buffer */
10     {     /* If space is not available, request a wakeup            */
11        mapwant(xx_map)++;  /* when space is returned                 */
12        sleep(xx_map, XX_MAPPRIO);  /* Wait for space; mfree          */
13                  /* will check mapwant and supply                  */
14                  /* the wakeup call.                          */
15     } /* endwhile */
16     splx(oldlevel);
```

Figure D3X–43 sleep Suspends Process Activity

When a request is made for a `cblock` (line 2), and the `cfreelist` is empty, set the `cblock` want flag. The driver waits for a free `cblock` (line 5), as shown below.

```
1       oldlevel = spl4();
2       while((cp = getcf()) == NULL)  /* Get free cblock from freelist */
3       {                              /* If freelist is empty,         */
4           cfreelist.c_flag++;        /* set cblock want flag          */
5           sleep(&cfreelist, TTIPRI); /* and wait for a free cblock    */
6       } /* endwhile */
7       splx(oldlevel);f1
```

Figure D3X–44 Driver Waiting for a Free cblock

In a driver **open**(D2X) routine, when a terminal device does not have carrier from a modem (line 24), the driver waits for carrier to be established (line 27), as shown below.

```
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) */
6                 /* and ring (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[]; /* Physical device register location */
11  extern struct tty    xx_tty[];  /* Logical device structure location */
12   ...
13  register struct tty *tp = &xx_tty[minor(dev)];
14  register struct device *rp = &xx_addr[minor(dev) >> 3]; /* Get device regs */
15  register int    port = minor(dev) & 0x07;       /* Get port number */
16   ...
17     oldlevel = spl6();
18     if  ((rp->modem_status & (0x0100 << port)) != 0) /* If carrier */
19     {                              /* to the modem            */
20        tp->t_state |= CARR_ON; /* indicate carrier established    */
21     } else {
22        tp->t_state &= ~CARR_ON;  /* else indicate carrier dropped */
23     } /* endif */

24     while((tp->t_state & CARR_ON) == 0)   /* While carrier not     */
25     {                        /* established, indicate driver    */
26          tp->t_state |= WOPEN; /* waiting for carrier            */
27          sleep((caddr_t)&tp->t_canq, TTIPRI); /* wait for carrier */
28     } /* endwhile */
29     splx(oldlevel);
```

Figure D3X–45 Driver Waiting for a Carrier

spl*(D3X)

NAME

spl — block/allow interrupts

SYNOPSIS

#include <sys/inline.h>

```
int oldlevel;
oldlevel = spl0();
oldlevel = spl1();
oldlevel = spl4();
oldlevel = spl5();
oldlevel = spl6();
oldlevel = spl7();
oldlevel = splhi();
oldlevel = splni();
oldlevel = splpp();
oldlevel = splstrm();
oldlevel = spltty();

splx(oldlevel)
int oldlevel;
```

ARGUMENT

oldlevel last set priority value (only **splx** has an input argument)

DESCRIPTION

When a process is executing code in a driver, the system will not switch context from that process to another executing process unless it is explicitly told to do so by the driver. This protects the integrity of the kernel and driver data structures. However, the system does allow devices to interrupt the processor and handle these interrupts immediately.

The integrity of system data structures would be destroyed if an interrupt handler were to manipulate the same data structures as a process executing in the driver. To prevent such problems, the kernel provides the **spl*** functions allowing a driver to set processor execution levels, prohibiting the handling of interrupts below the level set.

The selection of the appropriate **spl*** function is important. The execution level to which the processor is set must be high enough to protect the region of code; but this level should not be so high that it unnecessarily locks out interrupts that need to be processed quickly. A hardware device is assigned to one of two interrupt priority levels depending on whether it is a character device or a

block device. By using the appropriate **spl*** function, a driver can inhibit interrupts from its device or other devices at the same or lower interrupt priority levels.

When **sleep**(D3X) has been called to wait on the `cfreelist`(D4X) or for a `cblock`(D4X), use **spl4, spl5,** or **spltty**. When you are protecting a driver from interrupting itself, use **spl6** for block drivers, or **spl4, spl5**, or **spltty** for a character driver.

The specific interrupt priority levels (Table D3X-3) for each computer are (these values are defined in *ml/misc.s* and additionally for the 3B4000 computer and adjunct processors in *inline.h*).

Table D3X−4 spl Interrupt Priority Levels

spl*	SBC	3B2 and 3B4000 ACP	3B15 and 3B4000 MP	3B4000 EADP	Purpose
spl0	0	0	0	0	Allow all interrupts to be serviced
spl1	5	8	1	8	Mask context and process switch interrupts
spl4	8	10	7	11	Mask character device interrupts
spl5	10	10	7	11	Mask character device interrupts
spl6	16	12	10	11	Mask block device interrupts
spl7	15	15	15	15	Mask all interrupts
splhi	15	15	15	15	Mask all interrupts
splni†	12	12	10	--	Mask network interface interrupts
splpp†	10	10	--	11	Mask ports board interrupts
splstrm	--	8	8	8	Mask STREAMS device interrupts
spltty	13	13	7	7	Mask TTY device interrupts

†
Values for **splni** and **splpp** do not apply to the SBC.

The **spl*** command changes the state of the Processor Status Word (PSW). The PSW stores the current processor execution level, in addition to information relating to the operating system internals. The **spl*** commands block out interrupts that come in at a priority level at or below the interrupt priority level as shown in Table D3X-4.

The **spl*** functions are as follows

spl0 Restores all interrupts when executing on the base level. A driver routine may use **spl0** when the routine has been called through a system call; that is, if it is known that the level being restored is indeed at base level.

spl1 Used in context and process switch drivers to protect critical code.

spl4 Used in character drivers to protect critical code.

spl5 Used in character drivers to protect critical code (this function has the same effect as **spl4**.

spl6 Used in block drivers to protect critical code.

spl7 Used in any type of driver to mask out all interrupts including the clock, and should be used very sparingly.

splhi Used in any type of driver to mask out all interrupts including the clock, and should be used very sparingly. (This function is identical to **spl7**.)

splpp Used by drivers accessing a ports board to protect critical code. (This function is identical to **spl4** and **spl5**.)

spltty Used by a TTY driver to protect critical code.

splx Used to terminate a section of protected critical code. This function restores the interrupt level to the previous level specified by its argument *oldlevel*.

IMPORTANT: **spl*** functions should not be used in interrupt routines unless you save the old interrupt priority level in a variable as it was returned from an **spl*** call. Later, **splx** must be used to restore the saved old level. Never drop the interrupt priority level below the level at which an interrupt routine was entered. For example, if an interrupt routine is serviced at an interrupt priority level of 10 (**spl5**), do not call **spl0** through **spl4** or the stack may become corrupted.

RETURN VALUE

All **spl*** functions (except **splx**) return the former priority level.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 9, "Synchronizing Hardware and Software Events."

SOURCE FILE

ml/misc.s

EXAMPLE

If more then one driver routine (for example, base-level and interrupt-level routines) is permitted to manipulate system data structures at the same time, the integrity of the data can be destroyed. Or when a base level routine suspends the execution of the current process by calling the **sleep** routine, the driver must guarantee the **wakeup**(D3X) call does not take place before the call to **sleep** has completed.

To prevent such problems, the driver must make critical sections of code appear as one atomic operation. This is accomplished by not honoring (disable) interrupts at the appropriate level during the execution of critical sections of code.

Whenever a base level routine updates a system data structure that can also be updated by a interrupt level routine, the appropriate level of interrupts are disabled (line 3). The data structure is then updated (line 7), and the former interrupt levels are enabled (line 8), as shown in the next figure.

```
1    if (copyin(u.u_base, addr, size) == -1) /* Move to allocated buffer */
2    {                                       /* If invalid address found   */
3       oldlevel = spl4();       /* since an interrupt routine can also  */
4                   /* return buffers to the map, disable the          */
5                   /* appropriate level of interrupts before          */
6                   /* the buffer is returned to the map               */
7       mfree(xx_map, size, addr); /* Return buffer to management map   */
8       splx(oldlevel);            /* Enable former interrupt level     */
9       u.u_error = EFAULT;        /* Return error code                 */
10      return;
11   } /* endif */
```

Figure D3X–46 Enabling Interrupts

Whenever a driver makes a request for a system resource that may not be available, the driver first disables the appropriate level of interrupts (line 1) before the making the request (line 4). If the resource is not available, the driver sets the want flag and waits for the resource to become available (line 7). After the resource is made available, the driver enables the former interrupt levels (line 9), as shown in the following figure.

```
1  oldlevel = spltty(); /* Disable all interrupts through tty level.      */
2                  /* This ensures a test on the want flag cannot       */
3                  /* take place before the call to sleep has completed */
4  while((cp = getcf()) == NULL)      /* Get a free cblock from freelist */
5  {                              /* If freelist is empty, then      */
6      cfreelist.c_flag++;            /* set cblock want flag          */
7      sleep(&cfreelist, TTPRIO);     /* and wait for a free cblock    */
8  } /* endwhile */
9  splx(oldlevel);                    /* Return to old spl level       */
```

Figure D3X–47 Driver Enables Former Interrupt Levels

sptalloc(D3X)

NAME

sptalloc — allocate memory pages

SYNOPSIS

#include <sys/immu.h>

int
sptalloc(*size, mode, base, flag*)
int *size, mode, base, flag*;

ARGUMENTS

size the number of pages allocated

mode page descriptor table entry field mask. Possible values (defined in *immu.h*) are

- PG_ADDR physical page address
- PG_LOCK page lock bit (software)
- PG_NDREF need-reference bit (software)
- PG_REF reference bit
- PG_COPYW copy-on-write bit
- PG_LAST last-page bit
- PG_M modify bit
- PG_P page-present bit (the usual case)

base pointer to page descriptor entry (or entries). If multiple pages are being allocated, *base* is the pointer to the first entry. If *base* is NULL (the usual case), the system page descriptor entries that were setup to map the two megabytes of virtual space are used.

flag indicates whether the function allocating memory can call **sleep**(D3X). Valid nonzero values are NOSLEEP (defined in *immu.h*) and SE_NOSLP (defined in *stream.h*). Zero is also a valid value. When zero is set, the function can sleep to get memory. When a non-zero value is set, the function cannot sleep.

DESCRIPTION

This function allocates and links virtual memory pages for the 3B2 computer, 3B4000 computer, and the SBC. The normal return value is the kernel virtual address of the allocate space. Allocated space is virtually, but not physically contiguous (where its alignment is unimportant and small area are being allocated). **sptalloc** is much more efficient than **kseg**(D3X) (both functions allocate from the same map).

Except for page alignment, using **sptalloc** does not guarantee any alignment of allocated space.

On the SBC only, **sptalloc** is used to make certain types of peripherals such as graphics boards appear in the kernel virtual address space. Only SBC A24 peripherals will need to use **sptalloc**.

On the SBC, call **sptalloc** as follows:

vaddr `=sptalloc(btoc(`*size*`), (`*PG_P*`|`*PG_LOCK*`), btoc(`*paddr*`), 0);`

Where

vaddr a return address used to access the board

size the number of bytes to map (round up to a 2K boundary)

paddr physical address (create for the A24 board (4 bytes) by setting the high-order byte to 0x00 and the setting the low three bytes to the 3-byte A24 address)

*PG_** the mapping pages will be present and locked and the rest of the kernel does not have access to them.

0 the call will return failure immediately if the page tables are exhausted.

RETURN VALUE

Under normal conditions, the kernel virtual address of the allocated buffer is returned. Otherwise, NULL is returned when either virtual or physical memory cannot be allocated.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
kseg(D3X), **malloc**(D3X), **mapinit**(D3X), **mapwant**(D3X), **mfree**(D3X), **sptfree**(D3X), **unkseg**(D3X)

SOURCE FILE

os/page.c

sptfree(D3X)

NAME

sptfree — free allocated memory

SYNOPSIS

sptfree(*vaddr, size, flag*)
unsigned int *vaddr*;
int *size, flag*;

ARGUMENTS

vaddr base virtual address of memory to be released

size number of pages to be released

flag set to one to indicate memory should be freed (an area is to be released into the map). When *flag* is set to zero, it indicates that no memory is to be freed.

DESCRIPTION

This function releases memory or performs garbage cleanup to free allocated memory for re-use. This function is called after **sptalloc**(D3X) to free allocated memory.

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
kseg(D3X), **malloc**(D3X), **mapinit**(D3X), **mapwant**(D3X), **mfree**(D3X), **sptalloc**(D3X), **unkseg**(D3X)

SOURCE FILE

os/page.c

subyte(D3X) [OBSOLETE]

NAME

subyte — copy a byte from a driver to the user data space

SYNOPSIS

subyte(*userbuf, c*)
char **userbuf, c*;

ARGUMENTS

userbuf address of the user buffer

c byte to be copied

DESCRIPTION

The **subyte** function copies a byte from the driver buffer to user space.

When a driver **read**(D2X) or **write**(D2X) (not **ioctl**(D2X)) routine is entered, the **u.u_base** member of the `user`(D4X) structure contains the address of the buffer in the user address space, and the **u.u_count** member contains the number of bytes remaining to be transferred. After the **subyte** function completes, the driver should increase the value of the **u.u_base** member and decrease the value of the **u.u_count** member by the number of bytes transferred.

RETURN VALUE

subyte returns 0 (zero) if the transfer is successful. If a -1 is returned (an error occurred), set **u.u_error** to EFAULT to indicate that *userbuf* is a bad address.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

bcopy(D3X), **copyin**(D3X), **copyout**(D3X), **fubyte**(D3X), **fuword**(D3X), **iomove**(D3X), **suword**(D3X)

SOURCE FILE

ml/misc.s

EXAMPLE

Data can be moved between a `clist`(D4X) and a user data area one byte at a time. As long as there is space in the user data area, and there is data in the `clist`, obtain a single byte from the first `cblock`(D4X) in the `clist` (line 8) and copy it to the user data area (line 11).

```
1   extern struct tty xx_tty[];
2    ...
3   register struct tty *tp = &xx_tty[minor(dev)];
4   register int    c;
5    ...
6   while(u.u_count > 0)    /* While space in user data area  */
7   {
8       if  ((c = getc(&tp->t_canq)) == -1)  /* If input queue is empty, */
9           return;                     /* return                  */
10      /* endif */
11      if  (subyte(u.u_base++, c) == -1)    /* Copy character to user   */
12      {                               /* data area.  If invalid   */
13          u.u_error = EFAULT;             /* address is found, then   */
14          return;                     /* return error code        */
15      } /* endif */
16      u.u_count--;            /*  Update remaining size of data area */
17  } /* endwhile */
```

Figure D3X–48 Copying a Byte to User Data Space

suser(D3X)

NAME

suser — verify superuser permission mode

SYNOPSIS

suser()

DESCRIPTION

This function determines if the current user has superuser permissions.

RETURN VALUE

If the current user is superuser, 1 is returned. Otherwise, 0 (zero) is returned and **u.u_error** is set to EPERM (not owner).

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

useracc(D3X)

SOURCE FILE

os/fio.c

EXAMPLE

The use of **suser** is straight forward, easy to use, and viable for many situations. The following example shows such a test.

```
1   if (suser()==0)   /*Only superuser has access  */
2   {
3   return;           /*Return if permission denied */
4 }  /* endif */
```

Figure D3X−49 Determining if User is a Superuser

suword(D3X) [OBSOLETE]

NAME

suword — copy a word of data from a driver to user data space

SYNOPSIS

suword(*userbuf, i*)
int **userbuf, i*;

ARGUMENTS

userbuf address of the user buffer

i integer to be copied

DESCRIPTION

The **suword** function copies a single word from the driver buffer to user space.

When a driver **read**(D2X) or **write**(D2X) (not **ioctl**(D2X)) routine is entered, the **u.u_base** member of the `user`(D4X) data structure contains the address of the buffer in the user address space. The **u.u_count** member contains the number of bytes remaining to be transferred.

After **suword** completes, the driver should increase the value of the **u.u_base** member and decrease the value of the **u.u_count** member by the number of bytes transferred.

RETURN VALUE

suword returns a 0 (zero) if the transfer is successful. If a -1 is returned (an error occurred), set **u.u_error** to EFAULT to indicate that *userbuf* is a bad address.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

bcopy(D3X), **copyin**(D3X), **copyout**(D3X), **fubyte**(D3X), **fuword**(D3X), **iomove**(D3X), **subyte**(D3X)

SOURCE FILE

ml/misc.s

EXAMPLE

To debug a driver, a driver **ioctl** routine can be used to examine settings in the device registers such as the device status word. If a request is made for a device status word and the arg parameter contains a NULL pointer (line 19), return the value of the status word as the return code value of the **ioctl** system call (line 20). Otherwise, copy the value of the status word to the user data area specified by arg (line 23). If arg contains an invalid address, an error code is returned.

```
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 device xx_addr[]; /* Physical device register location */
9    ...
10  xx_ioctl(dev, cmd, arg, flag)
11  dev_t               dev;
12  caddr_t arg;
13  {
14  register struct device *rp = &xx_addr[minor(dev) >> 4];
15
16  switch(cmd)
17  {
18   case XX_GETSTATUS:
19            if  (arg  == NULL) {     /* If arg contains null pointer,*/
20               u.u_rval1 = rp->status; /* provide device status word */
21                        /*  as the return code from ioctl(2)  */
22                        /* system call                        */
23            } else if(suword(arg, rp->status) == -1) { /* Copy device */
24                         /* status word to user data area    */
25               u.u_error = EFAULT;  /* If invalid address found,    */
26               return;          /* then return error code       */
27            } /* endif */
28             break;
29          ...
```

Figure D3X–50 suword — Copies an Integer

timeout(D3X)

NAME

timeout — execute a function after a specified length of time

SYNOPSIS

timeout(*ftn, arg, ticks*)
int ((**void**)**ftn*)();
int*arg, ticks*;

ARGUMENTS

ftn kernel function to invoke when the time increment expires

arg argument to the function

ticks number of clock ticks to wait before the function is called

DESCRIPTION

The **timeout** function calls the specified function after a specified time interval. Control is immediately returned to the caller. This is useful when an event is known to occur within a specific time frame, or when you want to wait for I/O processes when an interrupt is not available or might cause problems. For example, some robotics applications do not provide a status flag for determining when to pump information to the robot's controller. By using **timeout**, the driver can wait a predetermined interval and then begin transferring data to the robot.

The exact time interval over which the timeout takes effect cannot be guaranteed, but the value given is a close approximation. The function called by **timeout** must adhere to the same restrictions as a driver interrupt handler. It can neither access the `user`(D4X) structure, nor use previously set local variables.

RETURN VALUE

Under normal conditions, an integer timeout identifier is returned (which may, in unusual circumstances, be set to 0). Otherwise, if the **timeout** table is full, the following panic message results:

```
PANIC:  Timeout table overflow
```

The **timeout** function returns an identifier that may be passed to the **untimeout**(D3X) function to cancel a pending request.

NOTE: No value is returned from the called function.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 9, "Synchronizing Hardware and Software Events."
delay(D3X), **iodone**(D3X), **iowait**(D3X), **sleep**(D3X), **ttywait**(D3X), **untimeout**(D3X), **wakeup**(D3X)

SOURCE FILE

os/clock.c, *os/adp_clock.c*

EXAMPLE

The following is an example of the **timeout** function.

```
timeout(xx_scan, 0, HZ/3); /* Schedule scan of modem status words */
                           /* in one third of a second            */
```

ttclose(D3X)

NAME

ttclose — close a TTY device

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    ttclose(tp)
    struct tty *tp;
```

ARGUMENT

tp address of the `tty`(D4X) structure associated with the device being closed

DESCRIPTION

The line discipline close function, **ttclose**, is called by the device driver **close**(D2X) routine.

The **ttclose** function dissociates the device from the process that opened it and resets the ISOPEN flag in the device internal state register (**tp->t_state**). **ttclose** calls **ttioctl** which calls the driver **proc**(D2X) routine with T_RESUME set to transmit any characters in the device transmit buffer (**tp->t_tbuf**) out to the terminal, clears out all the TTY buffers and queues, and returns to the `cfreelist`(D4X) all `cblock`(s) allocated to the device.

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
ttopen(D3X)

SOURCE FILE

io/tt1.c

EXAMPLE

On the last close of a terminal device, the driver **close**(D2X) routine 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` routine indirectly through the line discipline switch table (**l_close** is defined in *conf.h*) (line 6). The **t_line** member of the `tty` structure contains the line discipline (in this case 0 (zero)) and serves as the index to the line discipline switch table. After the logical data connection is terminated, the driver would break the physical connection (such as instructing the modem to drop carrier).

```
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 D3X–51 Data Connection is Terminated

ttin (D 3 X)

NAME

ttin — move a TTY character to the raw queue

SYNOPSIS

#include <sys/types.h>
#include <sys/tty.h>

ttin(*tp, code*)
struct tty **tp*;
int *code*;

ARGUMENTS

tp pointer to the `tty`(D4X) structure for a device

code [optional] set to L_BREAK if the BREAK key was entered. Upon receiving this *code*, **ttin** signals the processes identified by **t_pgrp** that the key was received and then calls **ttyflush** to release all buffers and wake up any processes sleeping on **t_outq, t_oflag,** and **t_rawq.**

DESCRIPTION

The **ttin** function works through the `tty` receive buffer to convert newline, carriage return, and uppercase characters and place them in the raw queue **t_rawq**. The mode members of the `tty` structure, define how these characters are converted.

The following paragraphs describe the highlights of what **ttin** performs. Refer to the Driver Development Guide appendix for further information.

If the number of characters in the raw queue exceeds the high water mark, **ttin** calls the driver **proc**(D2X) routine (with the T_BLOCK flag set) to send a stop character to the device. The high water mark is explained in the Glossary of this manual. When the raw queue character count exceeds the TTYHOG level of 256 characters, **ttin** calls **ttyflush** to flush the `tty` input queues. TTYHOG is defined in the *tty.h* header file of this manual. If the interrupt character (typically DELETE) or the quit character is found, **ttin** sends the appropriate signal to the process group associated with the device. If processes associated with the device are sleeping and **ttin** finds a line delimiter character, **ttin** awakens the sleeping process.

The **ttin** function also transmits characters to the terminal for display, if ECHO is enabled.

When the terminal operates in raw or non-canonical mode, the fifth and sixth elements of the `tty` structure control character array indicate the number of characters needed, and the amount of time waited before processes associated with the device should be awakened. If the minimum character count has been met, **ttin** awakens processes associated with the terminal.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
getc(D3X), **getcb**(D3X), **getcf**(D3X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **ttread**(D3X)

SOURCE FILE

io/tt1.c

EXAMPLE

When a driver is controlling a terminal device, it should use the TTY subsystem. This subsystem is a set of routines that provide terminal interface. Using the `clist`(D4X) and TTY data structures, the TTY subsystem provides both buffering and semantic processing of character data. All the information needed to perform I/O operations to a terminal is maintained in the `tty` structure. Therefore, a `tty` structure exists for every possible terminal device in the system.

After a driver receive interrupt routine validates an input character, it stores the character in the receive buffer (**t_rbuf**) (line 24). When the receive buffer is filled (line 25), it is added to the raw queue and a new receive buffer is allocated (**ttin**) (line 29). In order to allow other protocols, a driver must access the **ttin** routine indirectly through the line discipline switch table (**l_input** is defined in *conf.h*). The **t_line** member of the `tty` structure (line 29) contains the line discipline (in this case 0 (zero)) and serves as the index to the line discipline switch table.

```
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[];  /* Logical device structure location */
9   extern struct device xx_addr[]; /* Physical device register location */
10  extern int           xx_cnt;    /* Number of physical devices        */
11   ...
12  xx_rint(board)
13  int board;       /* The hardware board causing 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 valid data */
19  {                          /* in 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 stripped to   */
23  /*   proper bit size, character is stored in 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 c_ptr to first character in the receive buffer. The    */
27        /* driver must do operation to ensure the buffer added          */
28        tp->t_rbuf.c_ptr -= tp->t_rbuf.c_size; /* to raw queue correctly */

29        (*linesw[tp->t_line].l_input)(tp); /* Add receive buffer to     */
30        /* raw queue; get empty receive buffer                          */
31     } /* endif */
32  } /* endwhile */
33   ...
```

Figure D3X−52 ttin — Moves Character to Raw Queue

ttinit(D3X)

NAME

ttinit — initialize line discipline 0

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    ttinit(tp)
    struct tty *tp;
```

ARGUMENT

tp pointer to the `tty`(D4X) structure associated with the device being opened

DESCRIPTION

The TTY subsystem provides two functions, **ttinit**(D3X) and **ttopen**(D3X), for the driver **open**(D2X) routine. The driver calls **ttinit** function the first time a device is opened. **ttinit** resets the **t_line, t_iflag, t_oflag,** and **t_lflag** members of the `tty` data structure. It also sets the default control modes (**t_cflag**) and control characters (**t_cc**).

NOTE: **ttinit** is only usable for resetting line discipline 0. Use on any other line discipline requires resetting **t_line** to a new value after **ttinit** is called.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
open(D2X), **ttopen**(D3X)

SOURCE FILE

io/tty.c

EXAMPLE

When a driver **open** routine is called for a terminal device, the logical state of the device is checked. If the device has not previously been opened (ISOPEN) and is not currently being opened, the `tty` structure is initialized to its default values (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. The defaults from **ttinit** are 300 baud and 8 bit characters. These defaults enable receiver, and hang-up on last close.

```
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 not open */
12    {                        /* and waiting to be opened,          */
13       ttinit(tp);  /* initialize tty structure with default values   */
14       tp->t_proc = xx_proc;  /* Provide line discipline routines */
15    /* access to 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 D3X−53 Initializing tty Structure Default Values

ttiocom (D3X)

NAME

ttiocom — change device parameters

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>
#include <sys/termio.h>

    ttiocom(tp, cmd, arg, mode)
    struct tty *tp;
    int cmd, arg, mode;
```

ARGUMENTS

tp pointer to the `tty`(D4X) structure associated with the device to be controlled

cmd command regulates a device's input or output controls; refer to **termio**(7) for more information on the commands described here

Valid commands (listed in alphabetic order) are

TCSBRK	Waits for the output to drain. If *arg* is 0, then sends a BREAK character
TCFLSH	If *arg* is 0, flushes the input queue; if 1, flushes the output queue; if 2, flushes both the input and output queues.
TCGETA	Gets the parameters associated with the terminal and stores in the `termio` structure referenced by *arg*.
TCSETA	Sets the parameters associated with the terminal from the structure referenced by *arg*. The change is immediate.

TCSETAW — The same as TCSETA except that you wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.

TCXONC — Starts/stops control. If *arg* is 0, suspends output; if 1, restarts suspended output.

arg — Flag indicates which subordinate form of a command should be selected, or pointer to the `termio` structure associated with the device

mode — Contains the value of the **f_flag** member of the associated special device file (see *file.h*)

Note that the **ttiocom** function determines if an integer or an address is present in *arg* by the value of the *cmd* argument.

DESCRIPTION

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 require 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. That is, it is associated with the terminal protocol or line discipline.

RETURN VALUE

Under normal conditions, 0 (zero) is returned. Otherwise, 1 is returned to indicating the device registers must also be changed (1 is not an error code).

The following error values (set in **u.u_error**) are also possible:

EFAULT bad address. This value is set under the following conditions

TCGETA — **copyout** failed

TCSETA — **copyin** failed

EINVAL invalid argument. This value is set under the following conditions:

TCFLSH	*arg* not in the range of 0 to 2
TCSETA	line discipline value in the **c_line** member of the **termio** structure not in the range of 0 to 2
TCXONC	*arg* not in the range of 0 to 3

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
ioctl(D2X), **ttioctl**(D3X)

SOURCE FILE

io/tty.c

EXAMPLE

A 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 (line 16). 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. The line discipline switch table is **not** to be used for a line discipline 0 **ioctl** request.

NOTE: The **ttioctl** function is used internally with the other TTY subsystem routines for allocating and deallocating needed buffers, but provides nothing for the driver.

```
1   extern struct device xx_addr[]; /* Physical device register location */
2   extern struct tty    xx_tty[]; /* Logical device structure location */
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/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       /*  Changes usually determined by examining parameter          */
22       /*  settings in t_iflag, t_oflag, t_cflag, and t_lflag members */
23       /*  of tty structure for changes like baud rate, parity type,  */
24       /*  testing, etc. -- hardware dependent.                       */
25      } /* endif */

26     } /* endswitch */
27  } /* end xx_ioctl */
```

Figure D3X–54 Changing Device Parameters

ttioctl(D3X)

NAME

ttioctl — set device parameters

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>
#include <sys/termio.h>

    ttioctl(tp, cmd, arg, mode)
    struct tty *tp;
    int cmd, arg, mode;
```

ARGUMENTS

tp pointer to the `tty`(D4X) structure associated with the device controlled

cmd **ttioctl** *cmd*s are

- LDOPEN — allocates a receive buffer, a single `cblock`, to the **t_rbuf** character control block (`ccblock`), and calls the driver **proc** routine with the T_INPUT command so input can be initiated.
- LDCLOSE — resumes output by calling the driver **proc**(D2X) routine with the T_RESUME command, flushes the receive buffer (**t_rbuf**), and deallocates the `cblocks` assigned to the receive and transmit character control blocks (**t_rbuf** and **t_tbuf**).
- LDCHG — moves the entire character list of `cblocks` on the canonical queue to the raw queue if ICANON has been changed by a previous **ioctl** callin the **t_lflag** member of the `tty` structure.

arg flag indicates which subordinate form of a command should be selected, 0 is for LDOPEN and LDCLOSE. *arg* is the previous value of **t_lflag** if *cmd* is LDCHG.

mode contains the value of the **f_flag** member of the associated special device file (see *file.h*)

Note that **ttioctl** function determines if an integer or an address is present in *arg* by the value of the *cmd* argument.

DESCRIPTION

Changing the many parameters associated with terminal devices requires close cooperation between the driver and the TTY subsystem. The **ttioctl** 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 this.

Internally, **ttioctl** is called by **ttiocom**(D3X). These two functions both affect the appropriate parameter settings and return to the driver. **ttioctl** is specialized because it deals with parameters related to buffering and character processing. It is associated with the terminal protocol or line discipline.

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
ioctl(D2X), **ttiocom**(D3X)

SOURCE FILE

io/tt1.c

NAME

ttopen — open a TTY device

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    ttopen(tp)
    struct tty *tp;
```

ARGUMENT

tp pointer to the `tty`(D4X) structure associated with a device

DESCRIPTION

The TTY subsystem provides the **ttinit**(D3X) and **ttopen**(D3X) functions for the driver **open**(D2X) routine. The driver calls **ttinit** the first time a device is opened to set the `tty` structure to default values (including setting the line discipline to zero). The **ttopen** function is called each time the driver **open**(D2X) routine is called.

ttopen establishes the connection between the process and the device (**t_pgrp**). 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 **ttioctl** which calls the driver **proc**(D2X) routine with T_INPUT set.

RETURN VALUE

None. **ttopen** sets **t_state** to ISOPEN.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`linesw`(D4X), **open**(D2X), **ttclose**(D3X), **ttinit**(D3X)

SOURCE FILE

io/tt1.c

EXAMPLE

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 (see **ttinit**), the driver determines if a physical connection has been made by testing carrier from the modem (line 20). If a carrier is present (line 22), the `tty` structure indicates a physical connection has been made (line 24). Otherwise, the `tty` structure indicates a physical connection has not been made.

If the process wishes to wait for carrier, and carrier is not present, the driver waits for carrier (line 30). The last driver operation **open** routine is to establish a logical data connection and associate the device to a process by making the appropriate settings in the `tty` structure (line 33). In order to allow other protocols, a driver must access the **ttopen** routine indirectly through the line discipline switch table (**l_open** is defined in *conf.h*). The **t_line** member of the `tty` structure contains the line discipline (in this case 0 (zero)) and serves as the index to the line discipline switch table.

Interrupts are disabled during the **ttopen** call to insure all parameter settings in the `tty` structure are made before any testing and resetting of them is done by the driver interrupt and/or polling routines.

```
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)    */
6                   /* and ring (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[]; /* Physical device register location */
11  extern struct tty    xx_tty[]; /* Logical device structure location */
12   ...
13  xx_open(dev, flag)
14  dev_t dev;
```

Figure D3X–55 Opening a tty Device *(part 1 of 2)*

```
15  {
16  register struct tty *tp = &xx_tty[minor(dev)];
17  register struct device *rp = &xx_addr[minor(dev) >> 3]; /* Get device regs */
18   ...
19      oldlevel = spl6();
20      if  ((rp->modem_status & (0x0100 << port)) != 0) /* If a carrier */
21      {                                        /* to the modem */
22          tp->t_state |= CARR_ON;     /* indicate carrier established  */
23      } else {
24          tp->t_state &= ~CARR_ON;    /* else indicate carrier dropped */
25      } /* endif */

26      if  ((flag & FNDELAY) == 0) {   /* If process waits for carrier  */
27          while((tp->t_state & CARR_ON) == 0) /* while carrier not present */
28          {                       /* indicate process is waiting       */
29              tp->t_state |= WOPEN;  /* for carrier                    */
30              sleep((caddr_t)&tp->t_canq, TTIPRI);  /* Wait for carrier  */
31          } /* endwhile */
32      } /* endif */
33      (*linesw[tp->t_line].l_open)(tp); /* Establish logical connection    */
34      splx(oldlevel);
```

Figure D3X–55 Opening a tty Device *(part 2 of 2)*

ttout(D3X)

NAME

ttout — move a TTY character from **t_outq** to **t_tbuf**

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    ttout(tp)
    struct tty *tp;
```

ARGUMENT

tp pointer to the `tty`(D4X) structure associated with the device

DESCRIPTION

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 with T_TIME set to resume output. The **ttout** function also awakens a process which was asleep as a result of **ttwrite**(D3X) 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. *Low water mark* is defined in the Glossary of this manual.

RETURN VALUE

Under normal conditions, 0 (zero) is returned when there is no more data to process. Two special processing flags cause the CPRES value to be returned. (CPRES is set to octal 100000 in *tty.h*). The flags are

- EXTPROC is set by intelligent controller drivers to indicate that further processing will be handled by the controller (EXTPROC is defined in *tty.h*)
- OPOST is set by a driver (in the **t_oflag** of the `tty` structure) to indicate that output characters are post-processed as indicated by the other flags in the same structure member. (OPOST is defined in *termio.h* and described in the *Administrator's Reference Manual* under **termio**(7).)

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`linesw`(D4X), **ttin**(D3X)

SOURCE FILE

io/ttl.c

EXAMPLE

A driver transmit routine is entered when a device is ready to receive data (line 23). While the device is ready to receive data and the transmit register is free (line 25), get a character from the transmit buffer (**t_tbuf**) and place it in the transmit register (line 29). The state of the `tty` structure is changed to show a character is present in the transmit register (line 33) and the driver **proc** routine is called to complete the output (line 34).

The **proc** routine determines the output port (line 58). If output is blocked or there is no output for that port (line 59) then return. When the transmit buffer (**t_tbuf**) is empty, it is returned to the free list and a new transmit buffer is allocated from the output queue (line 61). The output character is transmitted to the device and the state of the `tty` structure is changed to show the transmit register is empty.

In order to allow other protocols, a driver must access the **ttout** function indirectly through the line discipline switch table (**l_output** is defined in *conf.h*). The **t_line** member of the `tty` structure contains the line discipline (in this case 0 (zero)) and serves as the index to the line discipline switch table.

```
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) */
6                 /* and ring (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[]; /* Physical device register location */
```

Figure D3X–56 A Driver Accesses ttout Function *(part 1 of 3)*

```
11  extern struct tty   xx_tty[]; /* Logical device structure location */
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;
20  port = rp->status & 0x7; /* Get terminal's port number          */
21  tp   = &xx_tty[(board << 3) & port]; /* Get tty structure       */
22  cp   = &tp->t_tbuf;                  /* Get transmit buffer    */

23  while((rp->status & XX_TXRDY) != 0) /* While 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 data in tbuf of the    */
28        {       /* tty structure, then give device the next     */
29           rp->xmit_char = *cp->c_ptr++; /* character for transmission */
30           cp->c_count--; /* update counter of 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 if output is blocked and if     */
35        /* not,  enable controller for transmission                   */
36     } else {
37        rp->control |= XX_TXDONE; /* Indicate data for port has been  */
38        break;                    /* transmitted; terminate loop      */
39     } /* endif */
40  } /* endwhile */
41  } /* end xx_xint */
```

Figure D3X−56 A Driver Accesses ttout Function *(part 2 of 3)*

```
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:

54  {
55     register struct ccblock *cp = &tp->t_tbuf;

56     if ((tp->t_state & (BUSY | TTSTOP)) != 0)   /* If no data to */
57        break; /* transmit or output blocked by <cntl>S, do nothing */

58     rp->xmit_char |= portmask; /* Enable controller to transmit character */

59     if (cp->c_ptr == NULL || cp->c_count == 0)  /* If no tbuf or */
60     {       /* tbuf empty, get a new one */
61        if ((*linesw[tp->t_line].l_output)(tp) & CPRES) == 0) /* If */
62           break;  /* no more output data, terminate output */
63     } /* endif */

64     tp->t_state |= BUSY; /* Indicate more output data in tbuf */
65                          /* and xmit_char register is clear  */
66     break;
67  }  /* end T_OUTPUT case */
68   ...
```

Figure D3X–56 A Driver Accesses ttout Function *(part 3 of 3)*

NAME

ttread — process an input TTY character

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    ttread(tp)
    struct tty *tp;
```

ARGUMENT

tp pointer to the `tty`(D4X) structure associated with the device from which the character is read

DESCRIPTION

The driver **read**(D2X) routine receives a device number as an argument. It uses this device number to determine the `tty` structure for the device being read. Then it uses the address of the `tty` structure as an argument to **ttread**.

After canonical processing, **ttread** transfers data from the canonical queue to user data space. If transmission from the terminal is blocked (**t_state** & TBLOCK) 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**(D2X) routine with T_UNBLOCK set to resume transmission from the terminal.

RETURN VALUE

Under normal conditions, no value is returned. Otherwise, **ttread** sets **u.u_error** to EFAULT if an error occurs when data is being transferred to the user data area. It is the driver's responsibility to check **u.u_error** when **ttread** is called.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
getc(D3X), **getcb**(D3X), **getcf**(D3X), `linesw`(D4X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **read**(D2X), **ttin**(D3X)

SOURCE FILE

io/tt1.c

EXAMPLE

When a process requests data from a terminal device, the driver **read** routine locates the `tty` structure associated with the device. The character data is copied from the input queues to the user data area (line 7). In order to allow other protocols, a driver must access the **ttread** function indirectly through the line discipline switch table (**l_read** is defined in *conf.h*). The **t_line** member of the `tty` structure contains the line discipline (in this case 0 (zero)) and serves as the index to the line discipline switch table.

```
1  extern struct tty   xx_tty[]; /* Logical device structures location */
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 input character data */
8                                         /* queues to user data area  */
9  } /* end xx_read */
```

Figure D3X–57 Processing an Input TTY Character

ttrstrt(D3X)

NAME

ttrstrt — restart TTY output after delay timeout

SYNOPSIS

ttrstrt(*tp*)
struct tty **tp*;

ARGUMENT

tp pointer to the `tty`(D4X) structure

DESCRIPTION

This function restarts TTY output following a delay timeout. **ttrstrt** calls the driver **proc** routine with the T_TIME flag set.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
timeout(D3X)

SOURCE FILE

io/tty.c

EXAMPLE

When a TCSBRK command is issued in a **ioctl**(2) system call, the line discipline routine **ttiocom**(D3X) calls the driver **proc** routine with the T_BREAK command (enters the **xx_proc** routine at line 33). The driver **proc** routine sends a break to the device (line 34). After the break is sent, output must be suspended for 250 milliseconds (HZ divided by 4). The **timeout**(D3X) function is used to call **ttrstrt** after the 250 milliseconds have elapsed (line 37). The **ttrstrt** function calls the driver **proc** routine with the T_TIME command so that output can be resumed (this call enters **xx_proc** at line 23). Refer to the following figure (lines 52 to 67) for the code for the **T_OUTPUT case** that is shown as comments in lines 29 and 30 of this example.

```
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) */
6                 /* and ring (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[];        /* Physical device registers */
11  extern struct tty   xx_tty[]; /* Logical device structures location */
12   ...
13  xx_proc(tp, cmd)              /* Driver command processing routine */
14  register struct tty *tp;
15  int cmd;
16  {
17  register int    dev = tp - xx_tty;  /* Compute minor device number  */
18  register struct device *rp = &xx_addr[dev >> 3]; /* Get device regs  */
19  register int    portmask = 0x0100 << (dev & 0x7);
20   /* Setup output port mask */
21  switch(cmd)
22  {
23  case T_TIME:   /* End timeout condition for T_BREAK                */
24      tp->t_state &= ~TIMEOUT; /* Indicate timeout condition completed */
25      goto resume_output;      /* Resume normal character output      */
26   ...
27  case T_OUTPUT: /* Perform output processing of data to the device     */
28  resume_output:
29              /* Transmit next tbuf character of the tty structure    */
30              /* See ttout reference page example program code        */
31      break;
32   ...
33  case T_BREAK:       /* Send a BREAK to a device                     */
34      rp->control |= XX_BRK;      /* Enable a break to be sent        */
35      rp->xmit_char |= portmask;  /* Enable controller & specify port */
36      tp->t_state |= TIMEOUT;     /* Show timeout condition in progress */
37      timeout(ttrstrt, tp, HZ/4); /* Disable timeout condition 1/4 of  */
38                                  /* a second (HZ) or 250 milliseconds */
39      break;
40   ...
```

Figure D3X–58 Restarts TTY Output After a Delay

NAME

tttimeo — time a character at a time terminal read request

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>
#include <sys/termio.h>

    tttimeo(tp)
    struct tty *tp;
```

ARGUMENT

tp pointer to the current `tty` structure

DESCRIPTION

This function times a character at a time terminal read request. A terminal may select to process characters a character at a time or a line at a time. Canonical processing is used on the latter. One method of handling characters that are received one at a time, is to set a time limit to wait until a character is received. This lets the program interpreting the input differentiate between characters keyed in and those that are transmitted by terminal protocol. The TIME constant defined in **termio**(7) provides more insight into timing data input.

The time limit is expressed in tenths of a second and is set in the constant **t_cc[VTIME]** variable of the `tty` structure. **tttimeo** is called by a subroutine set up to receive characters after **t_cc[VTIME]** tenths of seconds. After **tttimeo** is called, the caller must turn on IASLP in **t_state** and then call **sleep** using `(caddr_t)&tp->t_rawq` as the **sleep** event address and TTIPRI as the **sleep** priority.

tttimeo requires the following for input:

- RTO (timeout flag) must be disabled (in **t_state** in the `tty` structure)
- TACT (timeout in progress) must be set (in **t_state**)
- VTIME must be greater than zero
- ICANON must be disabled (in **t_lflag** of the `tty` structure)

tttimeo works by setting **t_state** to RTO and TACT, and then calling **timeout** to restart **tttimeo** in VTIME times HZ/10 ticks. When **tttimeo** is restarted, **t_state** is checked for RTO. If it is on, **t_state** is then checked for IASLP. If IASLP is on, **tttimeo** turns off IASLP in **t_state**, and wakes up any processes sleeping on the **t_rawq** raw input buffer.

RETURN VALUE

tttimeo returns prematurely if **t_state** is set to ICANON or **t_cc**[VTIME] is zero, or if **t_rawq.c_cc** is zero and **t_cc**[VMIN] is on (timing does not begin until the first character is input). If the system callout table is corrupted (and presumably the system in general), **timeout** panics the system. Upon completion, **t_delct** is set to 1.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
canon(D3X), **timeout**(D3X)

SOURCE FILE

io/tt1.c

EXAMPLE

The following example shows the use of **tttimeo** (line 14) in a terminal input routine.

```
1   /* line discipline input routine - transfer characters into rawq */
2   xxin(tp, code)
3   register struct tty *tp;
4   {
5       /* transfer characters into rawq from t_rbuf, doing any input
6       translations necessary at this point.  Echo character out to
7       outq if appropriate */
8       if (!(flg&ICANON)) {
9       tp->t_state &= ~RTO;
10          if (tp->t_rawq.c_cc >= tp->t_cc[VMIN])

11          tp->t_delct = 1;
12          else if (tp->t_cc[VTIME]) {
13              if (!(tp->t_state&TACT))
14                  tttimeo(tp);
15          }
16      }
17  }
```

Figure D3X–59 tttimeo Function

ttwrite(D3X)

NAME

ttwrite — move a TTY character from user address space to the output queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    ttwrite(tp)
    struct tty *tp;
```

ARGUMENT

tp pointer to the `tty`(D4X) structure associated with the device

DESCRIPTION

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.

A terminal driver's **write**(D2X) routine calls **ttwrite** to move the characters output from the user data space to the output queue. **ttwrite** also calls the driver's access routine to initiate actual output.

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 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**(D3X) to wait on the output queue. The **ttwrite** function calls the driver **proc**(D2X) routine with T_OUTPUT set to initiate or resume output to the device.

EXAMPLE

When a process requests data be transferred to a terminal device, the driver **write** routine locates the `tty` structure associated with the device. The data is copied from the user data area to the output queues (line 7) with a call through the line switch table `linesw`(D4X).

```
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);
8       /* Copy character data from user data area to output queues      */
9  } /* end xx_write */
```

Figure D3X−60 The ttwrite Function

RETURN VALUE

Under normal conditions, no value is returned. Otherwise, **ttwrite** sets **u.u_error** to EFAULT if an error occurs when data is being transferred from the user data area.

An EFAULT (bad address) error can be returned in **u.u_error** if the remaining characters cannot be written from user program space (**u.u_base**) to a `cblock`(D4X). This indicates that the `ublock` is corrupted, or that the `cblock` addresses are garbled.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`linesw`(D4X)

SOURCE FILE

io/tt1.c

ttxput(D3X)

NAME

ttxput — put characters into the TTY output buffer (**t_outq**)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    ttxput(tp, ucp, ncode)
    struct tty *tp;
    union {
        ushort ch;
        struct cblock *ptr;
    } ucp;
    int ncode;
```

ARGUMENTS

tp pointer to the `tty`(D4X) structure for the terminal being addressed

ucp either an unsigned **short** with the character to be output in the least significant byte, or a pointer to a `cblock`(D4X) structure containing the characters to be output on the terminal screen.

ncode set to zero if *ucp* is an unsigned **short**, or set to the number of characters to be output if *ucp* is a pointer to a `cblock`.

DESCRIPTION

This function transfers characters passed to it to the output queue, **t_outq**. **ttxput** also does output character translation if

- **t_state** does not have EXTPROC (external processing) on and **t_oflag** has OPOST set.
- **t_state** has EXTPROC set, but **t_lflag** has XCASE set. XCASE processing is always done in **ttxput** if EXTPROC is set.

ttxput places all characters passed to it into **t_outq**. In addition, if EXTPROC is not on and OPOST is set, **ttxput** performs the output processing described under the **t_oflag** member of the `tty` structure. This structure is documented under **termio**(7). This processing includes any translations of characters to the **t_outq** (for example, translating a "\n" to both "\n" and "\r"), and setting up for any delays necessary in outputting a special character like vertical tab, form feed, or carriage return. The delaying technique is then left to the line discipline output routine. **ttxput** places a QESC "character" into the **t_outq** followed by the actual character ORed with an octal 0200, if the character

is a delayed character. When processing a QESC character, the line discipline output routine should perform any appropriate delaying technique after outputting the character.

ttxput is called from any routine wishing to output a character to the terminal. The line discipline input routine calls **ttxput** to echo characters to the terminal if the ECHO bit of **t_lflag** is set. The line discipline write routine also calls **ttxput** to output characters to the terminal.

RETURN VALUE

ttxput returns after placing the character(s) onto the output queue with any appropriate translations defined by **t_oflag**.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
ttin(D3X), **ttwrite**(D3X)

SOURCE FILE

io/tt1.c

EXAMPLE

The following example uses **ttxput** (line 14) in a terminal input routine to echo characters to the terminal.

```
1   /* line discipline input routine - transfer
2    * characters to rawq from rbuf
3    */
4   xxin(tp, code)
5   register struct tty *tp;
6   {
7   register c;
8   cp = tp->t_rbuf.c_ptr;
9   c = *cp++;
10   /* transfer characters from t_rbuf to t_rawq performing input
11   translation if necessary */
12          if (flg&ECHO) {
13                /* place character  - 'c' - on t_outq */
14                ttxput(tp, c, 0);
15                /* initiate physical output */
16                (*tp->t_proc)(tp, T_OUTPUT);
17          }
18   /* check to see if non-canonical timing should be done */
19   }
```

Figure D3X−61 ttxput — Echoing Input Characters

ttyflush(D3X)

NAME

ttyflush — release TTY buffers

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    ttyflush(tp, rwflag)
    struct tty *tp;
    int rwflag;
```

ARGUMENTS

tp pointer to the `tty`(D4X) structure associated with the device

rwflag flag indicates whether use is in conjunction with a read or write operation. Valid values for this flag are FREAD and FWRITE.

DESCRIPTION

This function releases TTY buffers. If *cmd* is FREAD, **ttyflush**

1. releases the buffers in **t_canq** and **t_rawq** to the `cfreelist`(D4X)
2. calls the driver **proc**(D2X) routine with T_RFLUSH set
3. awakens any processes sleeping on **t_rawq**

If *cmd* is FWRITE, **ttyflush**

1. releases the buffers in **t_outq** to the `cfreelist`
2. calls the driver **proc** routine with T_WFLUSH set
3. awakens any processes sleeping on **t_outq**

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`cblock`(D4X) **clrbuf**(D3X)

SOURCE FILE

io/tty.c

ttywait(D3X)

NAME

ttywait — delay a process until character I/O operation is complete

SYNOPSIS

```
#include <sys/types.h>
#include <sys/tty.h>

    ttywait(tp)
    struct tty *tp;
```

ARGUMENT

tp pointer to the `tty`(D4X) structure associated with the device

DESCRIPTION

This function delays the execution of a process until the output of the Universal Asynchronous Receiver-Transmitter (UART) is drained (approximately 13 bit times, depending on baud rate). A UART is 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).

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
delay(D3X), **iodone**(D3X), **iowait**(D3X), **sleep**(D3X), **timeout**(D3X), **untimeout**(D3X), **wakeup**(D3X)

SOURCE FILE

io/tty.c

unkseg(D3X)

NAME

unkseg — free previously allocated kernel segment

SYNOPSIS

unkseg(*vaddr*)
char **vaddr*;

ARGUMENT

vaddr the starting virtual address of the memory to be released (returned by **kseg**(D3X))

DESCRIPTION

This function releases memory pages that were previously allocated by **kseg**(D3X). This function is not available on 3B2 swapping operating systems.

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
kseg(D3X), **malloc**(D3X), **mapinit**(D3X), **mapwant**(D3X), **mfree**(D3X), **sptalloc**(D3X), **sptfree**(D3X)

SOURCE FILE

os/mmgt.c

EXAMPLE

Some device controllers accept downloaded microcode. The preferred method of downloading microcode is to allocate a system buffer to hold the microcode. If the microcode is larger then a system buffer, break the code into segments the size of a system buffer and do repetitive moves to download the code. However, some device controllers may require all the microcode to be downloaded in a complete unit (segmentation is not permitted). Then the driver can dynamically allocate a private buffer using **kseg** (line 30), but the unit of allocation must be in pages or clicks which is converted with **btoc**(D3X). The microcode is copied to the allocated memory space (line 35). If an invalid address is found, an error condition is returned. Otherwise, the microcode is downloaded to the controller and the private buffer is deallocated (line 46).

```
1   struct  device   /* Physical device register layout */
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                       /* 0xf0; subdevice reporting in 0x0f  */
8           paddr_t   addr; /* Address of data to be read/written */
9           int       count; /* Amount of data to be read/written */
10  }; /* end device */
11  struct ucode         /* Layout of microcode input structure */
12  {
13          int       count;  /* Number of bytes in microcode    */
14          char      *code;   /* Location of the microcode       */
15  }; /* end ucode */
16  extern struct device *xx_addr[]; /*Physical register location  */
17  extern int           xx_cnt;    /* Number of devices */
18  ...
19  xx_ioctl(dev, cmd, arg, flag)
20  dev_t   dev;
21  caddr_t arg;
22  {
23  register struct device *rp;
24  switch(cmd)
25  {
```

Figure D3X−62 The unkseg Function *(part 1 of 2)*

```
26      case XX_DOWNLOAD:
27      {
28          register struct *ucp = (struct ucode *)arg; /* Get microcode */
29          register caddr_t bp;  /* location of private buffer */
30          if  ((bp = kseg(btoc(ucp->count)) == 0) /* Allocate buffer */
31          {   /* If insufficient memory for buffer space,  */
32              u.u_error = ENOMEM; /* return error condition */
33              return;
34          } /* endif */
35          if  (copyin(ucp->code, bp, ucp->count) == -1)  /* Copy microcode */
36          { /* to allocated buffer area; if invalid address found */
37            unkseg(bp);   /* Deallocate buffer area  */
38            u.u_error = EFAULT;   /* Return error condition */
39            return;
40          } /* endif */
41          rp = xx_addr[minor(dev) >> 3]; /* Get device registers */
42          rp->addr = vtop(bp, u.u_procp);  /* Set up the location */
43          rp->count = up_code.buffer_size; /* and size of microcode */
44          rp->control = XX_DOWNLD;      /* and download it       */
45          delay(HZ * 5);                /* Wait for completion   */
46          unkseg(bp);                 /* Deallocate buffer area */
47      } /* endblock */
48      break;
49      ...
```

Figure D3X–62 The unkseg Function *(part 2 of 2)*

NAME

untimeout — cancel previous **timeout**(D3X) function call

SYNOPSIS

untimeout(*id*)
int *id*;

ARGUMENT

id identification value generated by a previous **timeout** function call

DESCRIPTION

The **untimeout** function cancels a pending **timeout** request.

RETURN VALUE

Under any conditions, no value is returned on a 3B2 computer. Under normal conditions on an 3B4000 computer, 1 is returned. Otherwise on an 3B4000 computer, 0 (zero) is returned if the process associated with *id* is not found.

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 9, "Synchronizing Hardware and Software Events."
delay(D3X), **iodone**(D3X), **iowait**(D3X), **sleep**(D3X), **timeout**(D3X), **ttywait**(D3X), **wakeup**(D3X)

SOURCE FILE

os/clock.c, *os/adp_clock.c*

EXAMPLE

A driver may have to repeatedly request outside help from a computer operator. The **timeout** function is used to delay a certain amount of time between requests. However, once the request is honored, the driver will want to cancel the **timeout** operation. This is done with the **untimeout** function.

In a driver **open**(D2X) routine, 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. The driver schedules a wakeup call (line 41) and waits for 5 minutes (line 42). If the device is still not ready, the procedure is repeated.

When the device is made ready, an interrupt is generated. The driver interrupt handling routine notes there is a suspended process. It cancels the timeout request (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 I/O queue head     */
11        struct buf *mtu_tail;  /* Pointer to buffer I/O queue tail */
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 device registers */
17  extern struct mtu        mtu_tbl[]; /* Location of device structures */
18  extern int    mtu_cnt;
19   ...
20  mtu_open(dev, flag)
21  dev_t dev;
22  {
23  register struct mtu *dp;
24  register struct mtu_device *rp;
```

Figure D3X−63 The untimeout Function *(part 1 of 2)*

```
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;              /* return busy status   */
32          return;
33      } /* endif */
34      dp->mtu_flag = MTU_BUSY; /* Indicate device in use & clear flags */
35      rp = xx_addr[minor(dev) >> 3]; /* Get device regs */
36      oldlevel2 = splhi();
37      while((rp->status & MTU_LOAD) == 0) /* While tape 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 suspended  */
41          dp->mtu_to_id = timeout(wakeup, dp, 5*60*HZ); /* Wait 5 min */
42          if (sleep(dp, (PCATCH | PZERO + 2)) == 1) /* Wait on tape load */
43          {                   /* If user aborts process, release */
44              dp->mtu_flag = 0;  /* 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  } /* end mtu_open */
52   ...
53  mtu_int(cntr)
54  int cntr;       /* Controller that caused the interrupt */
55  {
56  register struct mtu_device *rp = xx_addr[cntr];  /* Get device regs */
57  register struct mtu *dp = &mtu_tbl[cntr << 3 | (rp->status & 0x3)];
58   ...
59      if ((dp->mtu_flag & MTU_WAIT) != 0)  /* If process is suspended */
60      {                           /* waiting for tape mount, */
61          untimeout(dp->mtu_to_id);     /* cancel timeout request  */
62          dp->flag &= ~MTU_WAIT;        /* Clear wait flag         */
63          wakeup(dp);                   /* Awaken suspend process  */
64      } /* endif */
65   ...
```

Figure D3X–63 The untimeout Function *(part 2 of 2)*

NAME

useracc — verify whether user has access to memory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/buf.h>

    useracc(base, count, access)
    caddr_t base;
    int count, access;
```

ARGUMENTS

base the start address of the user data area (the **u.u_base** member of the user structure).

count the size of the data transfer in bytes (for example, the **u.u_count** member of the `user`(D4X) structure).

access a flag to determine whether the access is a read or write. The defined constant B_READ specifies a write into memory (the user is performing a read operation). This requires that the user have write access permission for the specified data area. The defined constant B_WRITE specifies a read from memory. It requires read access permission for the data area. (B_READ and B_WRITE are defined in the system header file *buf.h*.)

DESCRIPTION

For raw I/O, a driver must verify that a user has access permission to the memory area specified in a **read**(D2X), **write**(D2X), or **ioctl**(D2X) system call. The kernel function **useracc** performs this verification. It is not necessary to use **useracc** for buffered I/O (including use of the **copyin**(D3X) and **copyout**(D3X) functions).

RETURN VALUE

Under normal conditions, 1 is returned. Otherwise, 0 (zero) is returned if the user does not have the proper access permission to the memory specified. If 0 is returned, set **u.u_error** to EFAULT.

LEVEL

Base Only (Do not call from an interrupt routine)

SEE ALSO

suser(D3X)

SOURCE FILE

os/probe.c

EXAMPLE

With a RAM disk, direct I/O requests can be handled in the driver **read** and **write** routines, as long as the I/O requests are for one or more complete blocks of information.

For either a **read**(2) or **write**(2) request, a test is made to determine if the I/O request is in the limits of the RAM disk (line 12). With a demand paging system, the driver must ensure that the user's program data pages are in memory (lines 19 and 48). If invalid pages are found, the driver returns the error condition if **useracc** has not already updated **u.u_error**.

```
1   #define RAMDNBLK   1000     /* RAM disk block number  */
2   #define RAMDBSIZ   512          /* Bytes per block     */
3   char ramdblks[RAMDNBLK][RAMDBSIZ]; /* Blocks forming RAM disk */
4   ramdread(dev)           /* Block device direct read request */
5   dev_t dev;
6   {
7   register daddr_t blkno; /* Starting block number */
8   register int     nblks; /* Blocks to be read (with direct (physio)*/
9    /* I/O from or to a block device, the data must be  */
10   /* moved as a single complete block or multiples of */
11   /* complete blocks).                                */
12   if (physck(RAMDNBLK,B_READ))  /* If read request in the limits */
13   {       /* of the RAM disk, copy data to user    */
14     if ((nblks = u.u_count / RAMDBSIZ)) <= 0) /* Determine */
15     {  /* number of blocks that can be copied.  If user data   */
16        u.u_error = EFAULT; /* area cannot hold complete block from*/
17        return;    /* the RAM disk, return error condition  */
18     } /* endif */
```

Figure D3X–64 The useracc Function *(part 1 of 3)*

```
19      if (useracc(u.u_base, u.u_count, B_READ) == 0)
20      {  /* Ensure process data pages are in core; */
21         if (u.u_error == 0) /* if not and u_error does not provide      */
22            u.u_error = EFAULT;  /* reason, return fault error condition */
23         return;
24      } /* endif */
25      blkno = u.u_offset / RAMDBSIZ;
26      /* Compute starting block number and copy data to user */
27       copy (u.u_base, (caddr_t)&ramdblks[blkno][0], (nblks * RAMDBSIZ));
28       u.u_base += i;    /* Increment virtual base addr */
29       u.u_offset += i;  /* Increment file offset */
30       u.u_count -= i;   /* Decrement bytes not transferred */
31    } /* endif */
32  } /* end ramdread */
33  ramdwrite(dev)    /* Direct write request to a block divide */
34  dev_t dev;
35  {
36  register daddr_t blkno;  /* Starting block number */
37  register int     nblks;  /* Number of blocks to be written; */
38    /* with direct I/O from or to a block device, the data must be  */
39    /* moved as a single complete block or multiples of */
40    /* complete blocks.                                 */
41       if (physck(RAMDNBLK,B_WRITE)) /* If write request in limits   */
42       {  /* of the ram disk, then copy data to user */

43          if (u.u_count % RAMDBSIZ != 0) /* Ensure there are one */
44          {  /* or more complete blocks to be copied. If not, then */
45             u.u_error = EFAULT; /* return an error condition.              */
46             return;
47          } /* endif */
```

Figure D3X–64 The useracc Function *(part 2 of 3)*

```
48          if (useracc(u.u_base, u.u_count, B_WRITE) == 0)
49          { /* Ensure process data pages are in core   */
50            if  (u.u_error == 0) /* If not and u_error gives no */
51                u.u_error = EFAULT;  /* reason, return fault error condition */
52            return;
53          } /* endif */
54          blkno = u.u_offset / RAMDBSIZ;
55          /* Compute starting block number and copy data to user */
56          copy (u.u_base, (caddr_t)&ramdblks[blkno][0], u.u_count);
57          u.u_base += i;  /* Increment virtual base addr */
58          u.u_offset += i;   /* Increment file offset */
59          u.u_count -= i;    /* Decrement bytes not transferred */
60        } /* endif */
61    } /* ramdwrite */
```

Figure D3X–64 The useracc Function *(part 3 of 3)*

vtop(D3X)

NAME

vtop — convert virtual to physical translation

SYNOPSIS

#include <sys/types.h>

```
paddr_t
vtop(vaddr, p)
char *vaddr;
struct proc *p;
```

ARGUMENTS

vaddr virtual address to convert

p pointer to the `proc`(D4X) structure used by **vtop** to locate the information tables used for memory management. To indicate that the address is in kernel virtual space or in the driver itself, set *p* to NULL. Block drivers that can transfer data directly in and out of user memory space must set *p* to the **b_proc** member of the `buf`(D4X) structure.

DESCRIPTION

This function converts a virtual address to a physical address. When a driver receives a memory address from the kernel, that address is virtual. Generally, memory management is performed by the CPU. However, devices that access memory directly (DMA), deal only with physical memory addresses. In such cases, the driver must provide the device with physical memory addresses.

The virtual address is the memory address being translated. The **vtop** function returns the translated address.

Note that **vtop** can only be used to translate a user address in the currently active process. As a result, it cannot be used in an interrupt handler or a time-out entry.

RETURN VALUE

Under normal conditions, a physical address is returned. Otherwise, the following can be returned:

- -1, if **vtop** is called with one argument and the physical address is incorrect.
- 0 (zero), if **vtop** is called with two arguments and the physical address is incorrect.

Should an error occur during address conversion, the following panic message is displayed:

```
PANIC: svirtophys - movtrw failed.
```

svirtophys is an internal function called by **vtop**. **movtrw** is an assembler instruction that handles the conversion between virtual and physical addresses.

LEVEL

If the *p* argument is set to NULL, **vtop** can be called from an interrupt routine. Otherwise, **vtop** can only be used in base level routines.

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
btoc(D3X), **ctob**(D3X), **getsrama**(D3X), and **getsramb(D3X)**

SOURCE FILE

os/machdep.c (3B2 computer) or *ml/misc.s* (3B4000 computer)

EXAMPLE

```
1   struct  mtu_device          /* Physical device register layout */
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 I/O queue head */
11   struct buf *mtu_tail;  /* Pointer to buffer I/O queue tail  */
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[];/* Physical device register location */
17  extern struct mtu        mtu_tbl[]; /* Logical device structure location */
18  extern int    mtu_cnt;
19   ...
20  mtu_iostart(rp, dp)
21  register struct mtu *dp;
22  register struct mtu_device *rp;
23  {
24  register struct buf *bp;
25  register int dev;

26     if (dp->mtu_flag & MTU_ACTIVE) != 0) /* If device is */
27        return; /* currently performing an I/O operation, return  */
28     /* endif */
29     if  ((bp = dp->mtu_head) == NULL)  /* Get next buffer in I/O queue  */
30         return;                    /* If none, return            */
```

Figure D3X−65 The vtop Function *(part 1 of 2)*

With direct memory access devices, the starting DMA address of a data area must be the physical address and not the virtual address, since the device does not have access to the virtual-to-physical translation (segment and/or page) tables. Therefore, at the start of an I/O operation, the driver provides the physical address plus the byte count. The device is then activated to perform the requested **read** or **write** operation (lines 40 to 46).

```
31      if (bp->b_flags & B_PHYS) != 0)   /* If direct transfer to user  */
32      {  /* program uses page tables associated */
33          rp->baddr = vtop(bp->b_un.b_addr, bp->b_proc) /* with process  */
34      } else {
35          rp->baddr = vtop(bp->b_un.b_addr, u.u_procp);
36          /* Buffer in kernel space, any page tables will suffice */
37      } /* endif */
38      rp->byte_cnt = bp->b_count; /* Number of bytes to transfer  */
39      dev = dp - mtu_tbl & 0x3;   /* Compute subdevice number     */
40    if (bp->b_flags & B_READ)    /* If a read request, then start */
41          rp->control = MTU_RD_CMD | dev; /* a read transfer of data */
42          else
43              rp->control = MTU_WR_CMD | dev; /* else start write data transfer */
44          /* endif */
45          dp->mtu_flag |= MTU_ACTIVE; /* Indicate device active with I/O request *
46  } /* end mtu_start */
```

Figure D3X–65 The vtop Function *(part 2 of 2)*

w a k e u p (D 3 X)

NAME

wakeup — resume suspended process execution

SYNOPSIS

#include <sys/types.h>

wakeup(*event*)
caddr_t *event*;

ARGUMENT

event unique address that is the same address used by **sleep**(D3X) to suspend process execution

DESCRIPTION

The **wakeup** function awakens all processes that called **sleep** with an address as the *event* argument. This lets the processes execute according to the scheduler. You must ensure that you use the same *event* for both **sleep** and **wakeup**. It is recommended for code readability and for efficiency to have a one-to-one correspondence between events and **sleep** addresses. Also, there is usually one bit in the driver flag member that corresponds to each reason for calling **sleep**.

Whenever a driver calls **wakeup**, it should test to ensure the *event* on which the driver called **sleep** occurred. There is an interval between the time the process that called **sleep** is awakened and the time it resumes execution where the state forcing the **sleep** may have been reentered. This can occur because all processes waiting for an event are awakened at the same time. The first process given control by the scheduler usually gains control of the event. All other processes awakened should recognize that they cannot continue and should reissue **sleep**.

RETURN VALUE

None

LEVEL

Base or Interrupt

SEE ALSO

BCI Driver Development Guide, Chapter 9, "Synchronizing Hardware and Software Events."
delay(D3X), **iodone**(D3X), **iowait**(D3X), **sleep**(D3X), **timeout**(D3X), **ttywait**(D3X), **untimeout**(D3X)

SOURCE FILE

os/slp.c

EXAMPLE

Sometimes a driver must suspend the execution of the current process with **sleep** while it waits for the availability of a hardware resource. It is the driver's responsibility to resume the suspended process with **wakeup** when the hardware resource is made available.

In a driver **open**(D2X) routine, when a terminal device does not have carrier from a modem, the driver waits for carrier to be established with **sleep**. The driver **scan** routine checks the status of the modems. (The **scan** routine (line 35) is a subordinate driver routine.) If a port is not open or is not waiting to be opened, skip it. When a modem carrier lead is on, but the `tty`(D4X) structure shows processes using the port waiting for carrier, awaken all suspended processes (line 46) and set the carrier-on flag (line 47).

```
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[]; /* Physical device register location */
11  extern struct tty    xx_tty[]; /* Logical device structure location */
12  ...
13  xx_open(dev, flag)
14  dev_t  dev;
15  int    flag;
```

Figure D3X–66 The wakeup Function *(part 1 of 2)*

```
16  {
17  register struct tty *tp = &xx_tty[minor(dev)];
18  register struct device *rp = &xx_addr[minor(dev) >> 3]; /* Get device regs */
19  register int    port = minor(dev) & 0x07;  /* Get port number */
20  ...
21      oldlevel = spl6();
22      if  ((rp->modem_status & (0x0100 << port)) != 0)
23      {  /* If carrier to the modem,       */
24         tp->t_state |= CARR_ON; /* Indicate carrier established  */
25      } else {
26         tp->t_state &= ~CARR_ON; /* else indicate carrier dropped */
27      } /* endif */

28      while((tp->t_state & CARR_ON) == 0)   /* While carrier is not */
29      {    /* established; indicate driver */
30           tp->t_state |= WOPEN;   /* Waiting for carrier */
31           sleep((caddr_t)&tp->t_canq, TTIPRI); /* Wait for carrier   */
32      } /* endwhile */
33      splx(oldlevel);
34      ...
35      xx_scan()
36      /* If port is open or */
37      /* Is waiting for carrier, then */
38            continue;      /* Skip this port */
39     /* endif */
40     rp = &xx_addr[port >> 3];      /* Get device registers */
41     mask = 0x0100 << (port & 0x7); /* Set up mask to test modem status word */
42     if  (rp->modem_status & mask) != 0)
43     {  /* If modem for port has carrier, */
44        if  ((tp->t_state & CARR_ON) == 0)   /* But processes waiting  */
45        {  /* for carrier, awaken   */
46           wakeup(&tp->t_canq);     /* All suspended processes    */
47           tp->t_state |= CARR_ON;  /* Show carrier established */
48        } /* endif */
49      ...
```

Figure D3X–66 The wakeup Function *(part 2 of 2)*

Refer to line 63 of the example for **untimeout**(D3X) for another **wakeup** example.

Section D4X: Data Structures (D4X)

Contents

Introduction

Section D4X describes the data structures used by drivers to share information between the driver and the kernel. All BCI driver data structures are identified with the (D4X) cross reference code.

Driver data structures are used as buffers for holding data passed between user data space and the device, as flags for indicating error device status, or as pointers to link buffers.

This section includes both general system data structures, such as the user area and the process table, and specific driver data structures, such as `buf` and `clist`. For ease of access, data structures are listed in alphabetical order.

The following structures are described:

- `bdevsw` contains system entry-points for block driver routines
- `buf` passes information between the block driver and the user program (also known as the buffer structure)
- The following structures are used together for buffering character data:
    - `cblock` accesses character data array
    - `ccblock` acts as temporary buffer for unqueued characters
    - `cfreelist` links a list of `cblocks`
    - `chead` heads the `cfreelist` pool of `cblocks`
    - `clist` passes information between most character drivers and the user program
- `cdevsw` contains system entry-points character driver routines
- D_FILE is a pointer used with the **drv_rfile**(D3X) function
- `hdedata` temporarily stores error information
- `iobuf` used to store private driver state information and to set up an internal queue for outstanding device I/O requests
- `linesw` contains entry points to the line discipline protocols for character driver processing and buffering
- `proc` process table structure locates the code, data, and stack information of a process. The scheduler also uses the `proc` structure in selecting processes to run.

- `sysinfo` indicates the number of times the **rint** and **xint** driver interrupt routines are entered
- TTY controls character transfers between a `tty` terminal driver and user data space
- `user` defines the process and its current state

IMPORTANT: The number of bytes in a structure may change at any time. Therefore, rely only on the structure members listed in this section and not on unlisted members or the position of a member in a structure.

On each manual reference page, each structure is organized by the following headings:

- DESCRIPTION provides general information about the structure
- STRUCTURE MEMBERS lists all accessible structure members
- SOURCE FILE indicates the path name location of the structure definition.
- SEE ALSO gives sources for further information

bdevsw (D4X)

DESCRIPTION

The `bdevsw` (block device switch table) data structure provides kernel entry points into a driver. `bdevsw` is constructed by the self-configuration process. `bdevsw` should not be used directly by a driver.

The `bdevsw` table allows the kernel to map the names of the devices to the device driver. It is used for block special files. The table includes pointers to functions used to implement user requests as shown in Figure D4X-1.

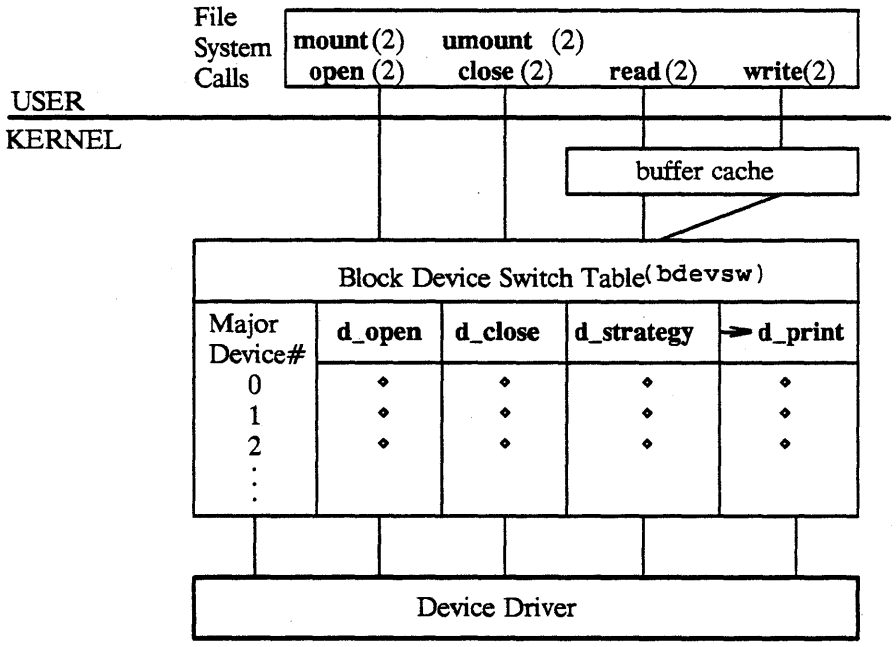


Figure D4X-1 bdevsw Structure

STRUCTURE MEMBERS

Type	Member	Description
int	(*d_open)();	/* Accesses driver open routine */
int	(*d_close)();	/* Accesses driver close routine */
int	(*d_strategy)();	/* Accesses driver strategy routine */
int	(*d_print)();	/* Accesses driver print routine */

SOURCE FILE

conf.h

SEE ALSO

BCI Driver Development Guide, Chapter 3, "Drivers in the UNIX Operating System."

buf(D4X)

DESCRIPTION

`buf` is the basic data structure for the system buffer cache used for block I/O transfers. Each buffer in the buffer cache has an associated buffer header. The header contains all the buffer control and status information. The buffer header pointer is the sole argument to a block driver **strategy**(D2X) routine. Do not depend on the size of the `buf` structure when writing a driver.

It is important to note that a buffer header may be linked in multiple lists simultaneously. Because of this, most of the members in the buffer header cannot be changed by the driver, even when the buffer header is in one of the driver's work lists.

Buffer headers are also used by the system for unbuffered or physical I/O for block drivers. In this case, the buffer describes a portion of user data space that is locked into memory.

In Figure D4X-2, two linked lists of buffers are illustrated. The top illustration is the `bfreelist`, the list of available buffers. The bottom illustration is a queue of allocated buffers. The lined areas indicate other buffer members.

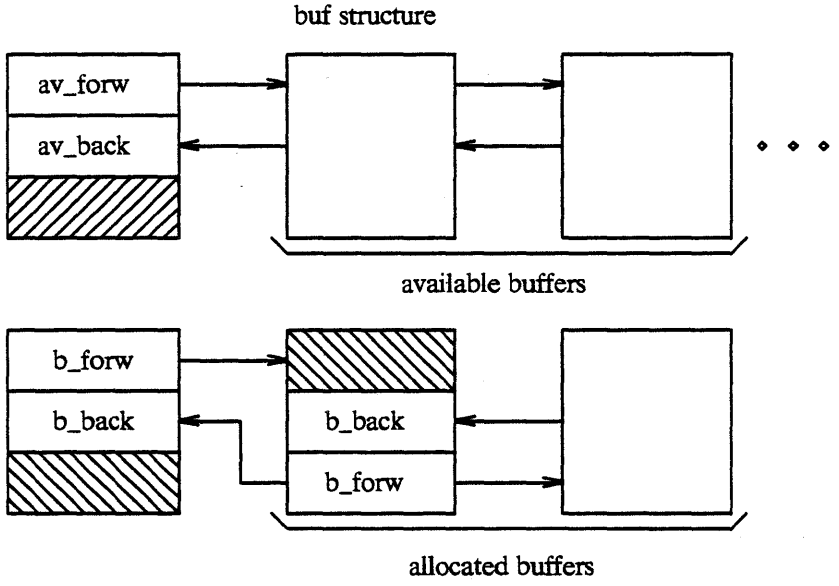


Figure D4X−2 buf Structure

STRUCTURE MEMBERS

Type	Member	Description
int	b_flags;	/* Buffer status */
struct buf	*b_forw;	/* Links the buffer into driver work lists */
struct buf	*b_back;	/* Links the buffer into driver work lists */
struct buf	*av_forw;	/* Position of buffer in free list */
struct buf	*av_back;	/* Position of buffer in free list */
dev_t	b_dev;	/* Major and minor device numbers */
unsigned	b_bcount;	/* Number of bytes to be transferred */
caddr_t	b_addr;	/* Buffer's virtual address */
daddr_t	b_blkno;	/* Logical block number */
char	b_error;	/* u.u_error error code number */
unsigned int	b_resid;	/* Number of bytes not transferred */
time_t	b_start;	/* I/O start time */
struct proc	*b_proc;	/* Process table entry address */

The **paddr** macro (defined in *buf.h*) provides access to the **b_un.b_addr** member of the `buf` structure. (**b_un** is a union that contains **b_addr**.)

The following example (Figure D4X-3) uses the **paddr** macro. The **paddr** macro is "passed" a pointer to a buffer header structure and "returns" the pointer to the buffer.

```
1   #include "sys/fs/s5param.h"

2   copy_the_data(bp)
3   struct buf *bp;
4   {

5   /* copy all the data from a buffer into user address space */

6   copyout(paddr(bp),u.u_base,SBUFSIZE);

7   }
```

Figure D4X−3 The paddr macro

Refer to Table D4X-1 for structure member field use.

Table D4X−1 buf Structure Member Use

Member	Use
b_flags	driver setable - Do not clear
b_forw	read only
b_back	read only
av_forw	read only
av_back	read only
b_dev	read only
b_bcount	read only
b_addr	read only
b_blkno	read only
b_error	driver setable
b_resid	driver setable
b_start	driver setable
b_proc	read only

CAUTION: The driver must never clear the **b_flags** member. If this is done, unpredictable results can occur including loss of disk sanity and the possible failure of other kernel processes.

The members of the buffer header available to test or set by a driver are as follows:

- **b_flags** stores the buffer status and tells the driver whether to read or write to the device. Valid flags are as follows:

  - B_BUSY — indicates the buffer is in use
  - B_DONE — indicates the data transfer has completed
  - B_ERROR — indicates an I/O transfer error
  - B_PHYS — indicates the buffer header is being used for physical (direct) I/O to a user data area. The **b_un** member contains the starting address of the user data area.

- B_READ — indicates data is to be read from the peripheral device into main memory
- B_WANTED — indicates the buffer is sought for allocation
- B_WRITE — indicates the data is to be transferred from main memory to the peripheral device. B_WRITE is a pseudo flag that occupies the same bit location as B_READ. B_WRITE cannot be directly tested; it is only detected as the "not" form of B_READ.

- **b_forw** and **b_back** can be used by the driver to link the buffer into driver work lists.
- **av_forw** and **av_back** maintain the position of the buffer on the buffer cache fre list.
- **b_dev** contains the external major and minor device numbers of the device accessed.
- **b_bcount** specifies the number of bytes to be transferred
- **b_un.b_addr** is the virtual address of the data buffer controlled by the buffer header. Data is read from the device to this starting address or is written to the device from this starting address.
- **b_blkno** identifies which logical block on the device (the device is defined by the minor device number) is to be accessed. The driver may have to convert this logical block number to a physical location such as a cylinder, track, and sector of a disk.
- **b_error** holds the error code that is eventually assigned to the **u.u_error** member of the `user` data structure by the kernel. It is set in conjunction with the B_ERROR flag (set by the operating system in the **b_flags** member).
- **b_resid** indicates the number of bytes not transferred because of an error.
- **b_start** holds the I/O operation start time. It can be used to measure device response time. See *BCI Driver Development Guide*, Chapter 14, "Performance Considerations."
- **b_proc** contains the process table entry address for the process requesting an unbuffered (direct) data transfer to a user data area (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.

geteblk Impact on the buf Structure

The **geteblk**(D3X) function allocates buffers. **geteblk** retrieves a buffer from the system buffer cache and returns the buffer header address to the calling routine. If a buffer header is not available, **geteblk** will sleep until one is available. When the buffer is obtained through **geteblk**(D3X), the driver may set the `buf` structure members as follows:

b_flags	set B_BUSY if the buffer is in use and set B_READ or B_WRITE depending on the transfer type (if a transfer was performed)
b_dev	set to the device number
b_bcount	set to the number of bytes in the buffer
b_blkno	set to the block on the device to be accessed
b_proc	set to 0 when the transfer is buffered

SOURCE FILE

buf.h

SEE ALSO

BCI Driver Development Guide, Chapter 6, "Input/Output Operations."
BCI Driver Development Guide, Chapter 4, "Header Files and Data Structures."
strategy(D2X), **physio**(D3X), **brelse**(D3X), **clrbuf**(D3X), **geteblk**(D3X) **iobuf**(D4X)

cblock(D4X)

DESCRIPTION

Character data is stored in an array which is part of a `cblock` structure. `cblocks` are linked together to form the `clist` (queue). `cblock` also contains indices to the first and last valid characters in the array.

The number of data characters in a `cblock` is set by the CLSIZE variable. The current value for CLSIZE is 64. Hence, a single `cblock` can contain up to 64 characters.

A `cblock` contains a pointer to the next `cblock` on a linked list (**c_next**), a small character array to contain data (**c_data**), and a set of offsets (**c_first** and **c_last**) indicating the position of the valid data in the `cblock` (see Figure D4X-4).

If there is not enough room in the `cblock` for all data, a new `cblock` is removed from the `cfreelist` and added to the end of the queue. If a `cblock` on a queue is empty, it is removed from the queue and placed on the `cfreelist`.

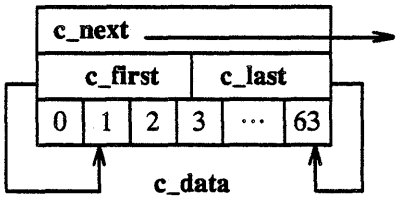


Figure D4X−4 cblock Structure

STRUCTURE MEMBERS

Type	Member	Description
struct cblock	*c_next;	/* Pointer to the next `cblock` */
char	c_first;	/* Index to the **c_data** array */ /* of the next character to be */ /* read from the `clist` */
char	c_last;	/* Index to the **c_data** array */ /* of the next character to be */ /* written to the `clist` */
char	c_data[CLSIZE];	/* `cblock` data */

SOURCE FILE

tty.h

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`ccblock`(D4X), `cfreelist`(D4X), `chead`(D4X), `clist`(D4X)

ccblock(D4X)

DESCRIPTION

A data structure used by the character I/O subsystem is the character control block, `ccblock`. `ccblock` is a temporary buffer for characters not in a queue.

The **c_ptr** member points to the character buffer (**c_data**) of a `cblock`. The **c_count** and **c_size** members are initialized to the size of the `cblock` character array (64 characters). The **c_count** member is then decreased by the number of characters in the `cblock` character buffer. The difference between the two members indicates the number of characters in the buffer.
See Figure D4X-5.

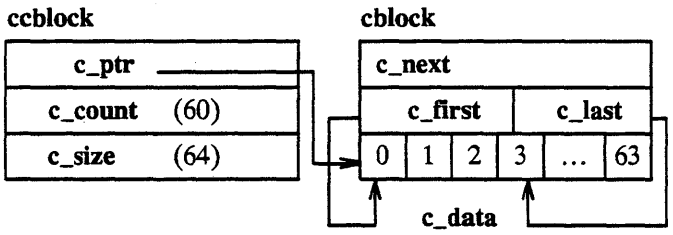


Figure D4X−5 ccblock Structure

The `ccblock` structure members are manipulated via the **t_tbuf** and the **t_rbuf** members of the `tty`(D4X) structure. For example, the following code example (see Figure D4X-6) accesses **c_count** and **c_size** members of the `ccblock` structure.

```
1   struct tty *tp;      /* tp is a pointer to the tty structure */
2   tp->t_tbuf.c_size -= tp->t_tbuf.c_count; /* Decrement c_size
3                                            by c_count      */
```

Figure D4X−6 Access in ccblock Structure

STRUCTURE MEMBERS

Type	Member	Description
caddr_t	c_ptr;	/* Buffer address */
ushort	c_count;	/* Character count */
ushort	c_size;	/* Buffer size */

SOURCE FILE

tty.h

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`cblock`(D4X), `cfreelist`(D4X), `chead`(D4X), `clist`(D4X)

cdevsw (D4X)

DESCRIPTION

The `cdevsw` (character device switch table) data structure provides driver entry points for the kernel. `cdevsw` is used for character special files. `cdevsw` is constructed by the self-configuration process. `cdevsw` should not be used by a driver. The structure members section illustrates how the switch table appears in memory and in the */unix* file.

The `cdevsw` table allows the kernel to map the names of devices to the device driver (see Figure D4X-7). The table includes pointers to functions used to implement user requests.

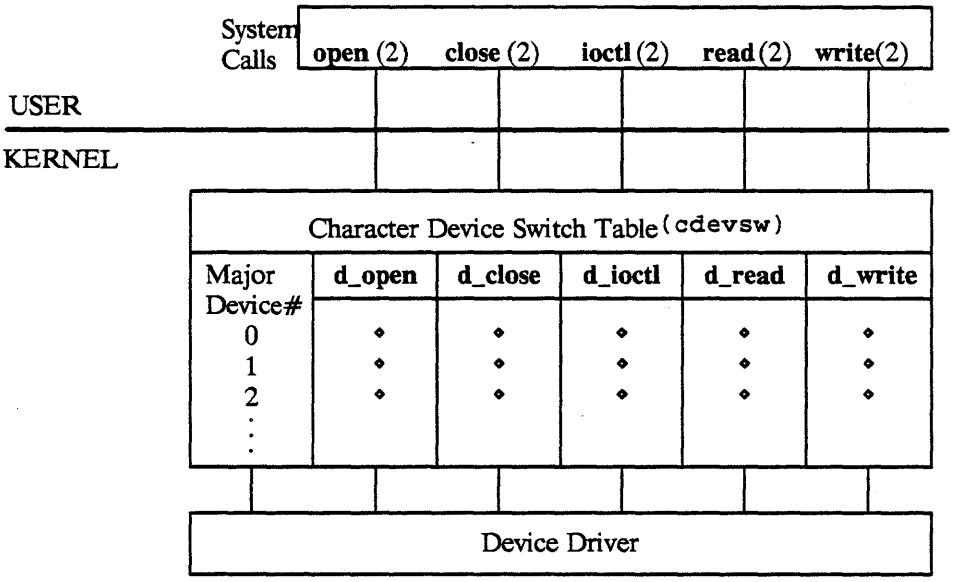


Figure D4X−7 cdevsw Structure

STRUCTURE MEMBERS

Type	Member	Description
int	(*d_open)();	/* Accesses driver open routine */
int	(*d_close)();	/* Accesses driver close routine */
int	(*d_read)();	/* Accesses driver read routine */
int	(*d_write)();	/* Accesses driver write routine */
int	(*d_ioctl)();	/* Access driver ioctl routine */

SOURCE FILE

conf.h

SEE ALSO

Section D2X, "Driver Routines (D2X)," of this manual.
BCI Driver Development Guide, Chapter 2, "Drivers in the UNIX Operating System."

cfreelist(D4X)

DESCRIPTION

cblocks are drawn from the cfreelist pool. cfreelist is headed by the chead data structure whose members are listed on this page. The size of cfreelist is determined by the NCLIST tunable parameter defined in the *kernel* master file.

The **c_flag** member indicates a process is waiting for a cblock. When a character device needs a cblock and the cfreelist is empty, the device must set the **c_flag** member of the chead structure to a non-zero value and **sleep** on the address of the cfreelist.

The cfreelist is a singly linked list (**c_next**) of cblocks(D4X). (See Figure D4X-8.) The **c_size** variable in the clist head structure indicates the size of the cblock character buffer. Since the cfreelist is limited in size and shared by all TTY devices, it is possible for the cfreelist to be empty when a cblock is needed by a TTY device. When this occurs, the process needing a cblock must use **sleep**(D3X) to wait on the cfreelist (see Figure D4X-9). This must not be done from the driver's interrupt level.

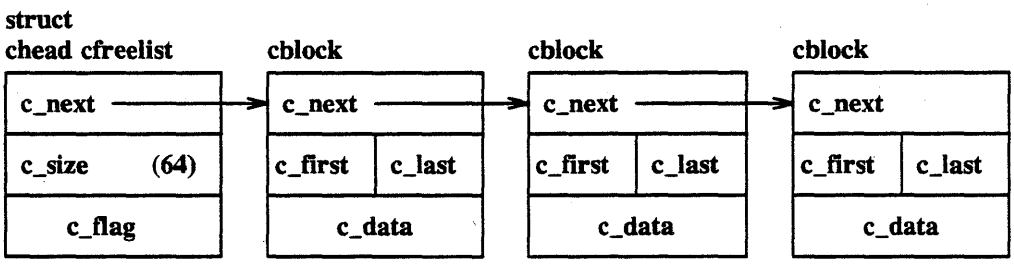


Figure D4X-8 cfreelist Structure

```
1    cfreelist.c_flag = 1;
2    sleep(&cfreelist, tty_priority);
```

Figure D4X-9 Waiting for an Available cfreelist Buffer

STRUCTURE MEMBERS

Type	Member	Description
struct cblock	*c_next;	/* Singly linked list */
int	c_size;	/* Size of the cblock character buffer */
int	c_flag;	/* Indicates process waiting for a cblock */

SOURCE FILE

tty.h

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`cblock`(D4X), `ccblock`(D4X), `chead`(D4X), `clist`(D4X)

chead (D4X)

DESCRIPTION

The pool from which cblocks(D4X) are drawn is the `cfreelist`. The `chead` structure points to the beginning of the `cfreelist`.

`cfreelist` is a singly linked list of `cblocks`. **c_next** points to the next available `cblock` in the queue. The **c_size** indicates the `cblock` character buffer size. There is a maximum of 64 bytes. **c_flag** indicates if a process is waiting for a `cblock`. (See Figure D4X-10.)

When a `cblock` is needed, an available `cblock` is removed from the `cfreelist` and added to the end of the queue. Since the `cfreelist` is limited in size and shared by all TTY devices, it is quite possible for `cfreelist` to be empty when a `cblock` is needed by a TTY device. When `cfreelist` is empty, the device must set **c_flag** to a non-zero value and sleep on the address of the `cfreelist`.

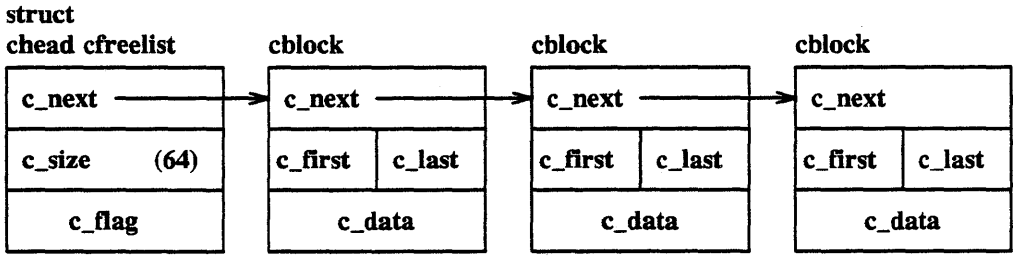


Figure D4X−10 chead Structure

STRUCTURE MEMBERS

Type	Member	Description
struct cblock	*c_next;	/* Singly linked list */
int	c_size;	/* Size of the cblock character buffer */
int	c_flag;	/* Indicates process waiting for a cblock */

SOURCE FILE

tty.h

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`cblock`(D4X), `ccblock`(D4X), `cfreelist`(D4X), `clist`(D4X)

clist (D4X)

DESCRIPTION

Character I/O is usually buffered in data structures that form a linked list queue called a character list, or `clist`. The `clist` is the head of a linked list queue of `cblocks`. It stores small quantities of data shared between a device and a user data area.

Typically, the terminal speed sends data at a slower rate than data can be sent to the user program. A character driver accumulates characters from the terminal in a `clist` and then passes the data to the user program.

`clist` contains a total count on the number of characters in the queue (**c_cc**) and pointer to the first (**c_cf**) and last (**c_cl**) `cblocks` in the queue. The `cblocks` form a singly linked list (**c_next**). Each `cblock` contains a buffer of up to 64 characters (**c_data**) and maintain indexes which point to the first (**c_first**) and last (**c_last**) character in the buffer.

This `clist` structure contains 172 bytes. This number is indicated by the value in c_cc member (See Figure D4X-11).

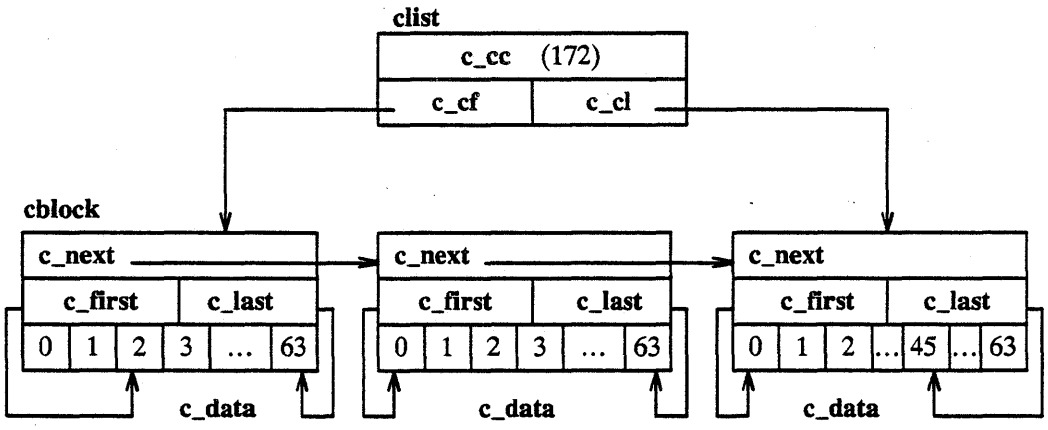


Figure D4X–11 clist Structure

STRUCTURE MEMBERS

Type	Member	Description
int	c_cc;	/* Number of Characters in the clist */
struct cblock	*c_cf;	/* Pointer to the first cblock */
struct cblock	*c_cl;	/* Pointer to the last cblock */

SOURCE FILE

tty.h

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`cblock`(D4X), `ccblock`(D4X), `cfreelist`(D4X), `chead`(D4X)

D_FILE(D4X) [3B15 and 3B4000 computers only]

DESCRIPTION

The `D_FILE` data structure is a pointer used in conjunction with the **drv_rfile**(D3X) function. **drv_rfile** reads a file into a buffer that it creates. The buffer address and buffer size are returned.

drv_rfile is useful for bringing a file into a driver, and for accessing files pumped (downloaded) to an intelligent controller.

drv_rfile should be called twice, once to open and read the file, and again to close the file. When the file is closed, the buffer is released.

IMPORTANT: Before **drv_rfile** is called, the name of the file read must reside in kernel space; not user space. `D_FILE` points to the complete path name of the file read, the entries where **drv_rfile** writes the buffer address, the buffer size, and if the file should be opened or closed. **drv_rfile** cannot be used in a driver's **init** routine. (It can be used in the **start** routine.)

Before calling **drv_rfile** with the open flag set, set **buffer_address** to NULL. This field must not be altered between open and close requests. Once **drv_rfile** has been given an open request, **drv_rfile** must close the request when completed to ensure that the buffer is freed correctly. In addition, the **open_close** flag should not be changed between the open and close calls of **drv_rfile**.

drv_rfile returns (-1) if an error occurred; it also sets **u.u_error** as follows:

ENOENT no file name or path too long

EFAULT cannot copy filename to internal buffer

ENOMEM cannot acquire buffer space as **drv_rfile** uses **kseg**(D3X)

EIO Read of file failed. **drv_rfile** will clean up. There is no need to issue a second call to close a file.

NOTE: **u.u_error** may have additional values if **drv_rfile** returns -1 because it called another kernel routine. For a `D_FILE` code example, see the **drv_rfile**(D3X) reference page in this manual.

STRUCTURE MEMBERS

Type	Member	Description
char	*file_name;	/* Name of file accessed */
char	*buffer_address;	/* Buffer address set to zero before open */
int	buffer_size;	/* Buffer size set to NULL before open */
char	open_close;	/* Open or close flag. open = 0, close = 1 */

SOURCE FILE

systm.h

SEE ALSO

drv_rfile(D3X), **kseg**(D3X)

hdedata(D4X)

DESCRIPTION

The `hdedata` data structure temporarily stores hard disk error information sent to an error queue. An `hdedata` structure is initialized for every disk on the system by driver calls to **hdeeqd**(D3X) when the system is booted. An error queue is also initialized by **hdeeqd**.

When the disk driver finds an error, it calls the **hdelog**(D3X) function with the error information. **hdelog** passes the `hdedata` structure for the error to the error queue. This error queue is a queue of bad block reports that have not been remapped. This queue resides in the kernel and not on the disk.

After a number or errors are accumulated, an administrator examines the list of errors collected in the queue. If any of the errors need to be "fixed", he or she remaps the bad block. Remapping means that the block address is rewritten to a defect table on the disk. PD sector information points to this defect table.

See the *BCI Driver Development Guide*, Chapter 11, "Error Reporting", for further information.

Figure D4X-12 illustrates the logging of hard disk errors.

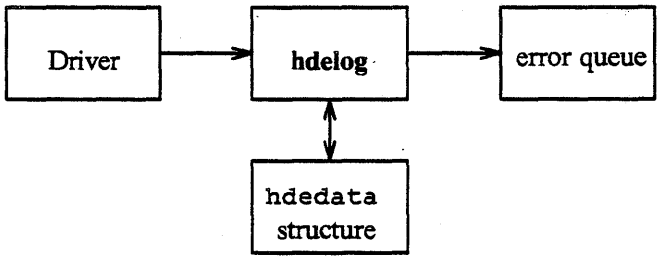


Figure D4X−12 Logging a Queue Error

When an error is encountered, the following message appears on the console:

`WARNING:` *severity readtype* `hard disk error:`
`maj/min =` *external-major-num, external-minor-num*

STRUCTURE MEMBERS

Type	Member	Description
dev_t	diskdev;	/* Major/minor disk device number */ /* (major number for character device) */
char	dskserno[12];	/* Disk pack serial number (can be all zeros) */
daddr_t	blkaddr;	/* Physical block address */ /* in machine-independent form */
char	readtype;	/* Error type: CRC (cyclical redundancy check */ /* or ECC(error check and correction) */
char	severity;	/* Severity type: marginal or unreadable */
char	badrtcnt;	/* Number of unreadable tries */
char	bitwidth;	/* Bitwidth of corrected error: 0 if CRC */
time_t	timestmp;	/* Time stamp */

The disk pack serial number is not currently evaluated, but it must contain a value. Set to all zeros.

SOURCE FILE

hdelog.h

SEE ALSO

BCI Driver Development Guide, Chapter 11, "Error Handling and Reliability."
hdeeqd(D3X), **hdelog**(D3X)

iobuf(D4X)

DESCRIPTION

The `iobuf` structure provides a template for a private I/O queue to manage a specific device's outstanding I/O requests, and fields to store device state information. The `iobuf` structure stores such information as the device number, an error count, the device's local bus address, and provides pointers to the **av_forw** and **av_back** fields in the `buf` structure. These pointers can be used to create an internal request queue.

3B15 computer and 3B4000 master processor IDFC device controllers use the `iobuf` structure specifically. Each IDFC controller has an `iobuf` structure, which contains private state data and two list heads: the **b_forw/b_back** list and the **d_actf/d_actl** list. The **b_forw/b_back** list is doubly linked and has all the buffers currently associated with that major device. The **d_actf/d_actl** list is private to the controller but is always used for the head and tail of the I/O queue for the device. Various routines in *bio.c* look at **b_forw/b_back** (notice they are the same as in the `buf` structure) but the rest is private to each device controller.

STRUCTURE MEMBERS

Type	Member	Description
int	b_flags;	/* see buf.h */
struct	buf *b_forw;	/* first buffer for this dev */
struct	buf *b_back;	/* last buffer for this dev */
struct	buf *b_actf;	/* head of I/O queue (b_forw)*/
struct	buf *b_actl;	/* tail of I/O queue (b_back)*/
dev_t	b_dev;	/* major+minor device name */
char	b_active;	/* busy flag */
char	b_errcnt;	/* error count (for recovery) */
int	jrqsleep;	/* process sleep counter on jrq full */
struct	eblock *io_erec;	/* error record */
int	io_nreg;	/* number of regs to log on errors */
paddr_t	io_addr;	/* local bus address */
physadr	io_mba;	/* mba address */
struct	iostat *io_stp;	/* unit I/O statistics */
time_t	io_start;	
int	sgreq;	/* SYSGEN required flag */
int	qcnt;	/* outstanding job request counter */
int	io_s1;	/* space for drivers to leave things */
int	io_s2;	/* space for drivers to leave things */

SOURCE FILE

iobuf.h

SEE ALSO

BCI Driver Development Guide, Chapter 4, "Header Files and Data Structures."
buf(D4X)

linesw (D4X)

DESCRIPTION

Line discipline is a term describing input/output character interpretation between the operating system and a terminal. It is the method by which characters are processed as they are sent and received from a terminal. The routines called by each attribute of a line discipline manipulate data in `clists`(D4X). The routines in `linesw` are invoked by the terminal driver.

Line refers to the phone line or cable that connects the character device to a controller. *Discipline* refers to the rules for character processing. Line discipline modules are called by terminal drivers to handle interactive use of the UNIX operating system. (See `tty`(D4X) for a diagram.) The functions of a line discipline are as follows:

- forms lines from input strings
- processes erase and kill characters (typically, backspace and at sign) causing previously entered information to be erased
- echos received characters to the terminal
- handles output character processing, including tab expansion.
- sends signals when the phone is hung up, the line is broken, or when a character such as DEL (delete) causes a process to stop.
- includes a raw (transparent) mode so characters can be sent directly from terminal to user process without any input processing.

`linesw` is an internal table containing a list of the routines supported for each line discipline.

Figure D4X-13 illustrates how `linesw` translates a request for a line discipline function into a request for a **tt***(D3X) function.

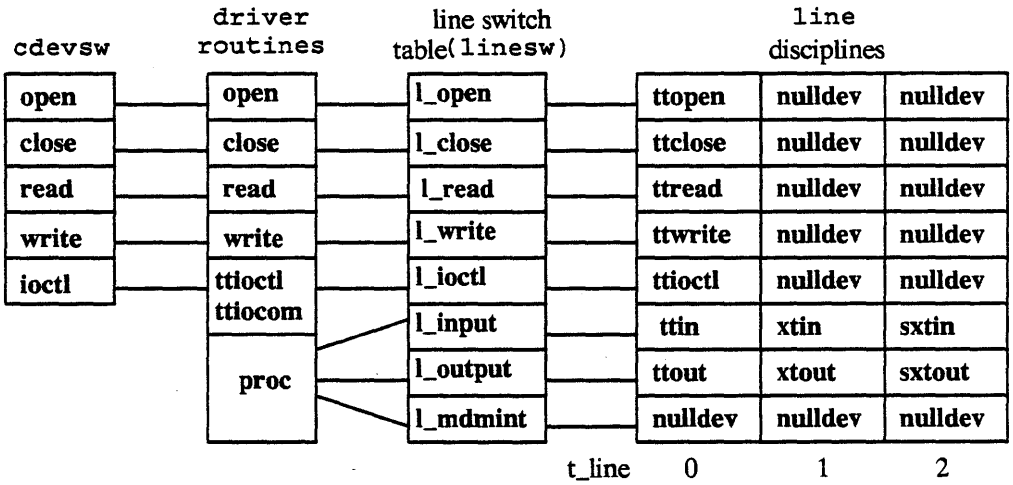


Figure D4X–13 linesw Structure

Valid line discipline values are 0, 1, and 2. These values represent

- Line discipline 0 is the TTY driver standard value
- Line discipline 1 is a special protocol for AT&T bit-mapped graphics terminals, such as the AT&T 630
- Line discipline 2 is used with **shl**(1), the shell **layers**(1) command

The TTY routines comprise the default, system-supplied line discipline, and line discipline (zero) (the first entry in the `linesw`). To allow other protocols, drivers must access the TTY routines indirectly through the line discipline switch table. The **t_line** member of the `tty` structure indexes the line discipline switch table.

There are eight members in the `linesw` structure. Each member handles a different attribute of character processing between a character driver and a terminal. The **l_mdmint** member provides for a modem interrupt handler, but is not presently used. This member is made non-functional by containing the address to the **nulldev**(D3X) function.

STRUCTURE MEMBERS

Type	Member	Description
int	(*l_open)();	/* Starts access to a terminal */
int	(*l_close)();	/* Discontinues access to a terminal */
int	(*l_read)();	/* Reads information from a terminal */
int	(*l_write)();	/* Writes information to a terminal */
int	(*l_ioctl)();	/* Handles I/O control functions */
int	(*l_input)();	/* Handles input interrupts */
int	(*l_output)();	/* Handles output interrupts */
int	(*l_mdmint)();	/* Handles modem interrupts */

The `linesw` structure is initialized by the **lboot** program as shown (see Figure D4X-14).

```
1   linesw [ ] = {
2   ttopen, ttclose, ttread, ttwrite, ttioctl, ttin, ttout, nulldev,
3   0
4   } ;
```

Figure D4X–14 linesw Initialization

SOURCE FILE

conf.h

SEE ALSO

Section D3X, "Kernel Functions (D3X)," in this manual.
BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."

proc(D4X)

DESCRIPTION

Each process is allocated a `proc` (process table) data structure containing the information defining the process and its state to the kernel. The `proc` structure contains required kernel information pointing to storage outside the kernel (see Figure D4X-15).

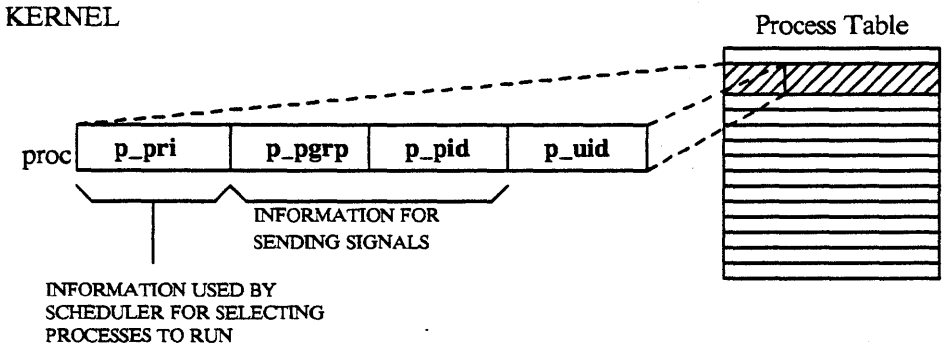


Figure D4X−15 proc Structure

Since `proc` structures are defined by the kernel, they are subject to change from one software release to another.

Since some drivers require access to certain fields of this structure, the following fields are not subject to change:

STRUCTURE MEMBERS

Type	Member	Description
char	p_pri;	/* The CPU priority of a process used by the */
		/* scheduler determines which process gets to execute. */
short	p_pgrp;	/* Process group identification number */
		/* determines which processes should receive a */
		/* HANGUP or BREAK signal, detected by a driver. */
short	p_pid;	/* Process identification number */
ushort	p_uid;	/* Process user id */

CAUTION: A driver should never modify this structure directly.

SOURCE FILE

proc.h

sysinfo(D4X)

DESCRIPTION

The `sysinfo` (system information) data structure is used by character drivers with the **rint**(D2X) and **xint**(D2X) driver interrupt routines. Drivers use the structure, which is accessed through code, to indicate the number of times **rint** or **xint** are entered. There are other member in this structure, but they are not used directly by the driver.

When a hardware or software interrupt occurs, `sysinfo` members are increased depending on the type of interrupt (see Figure D4X-16).

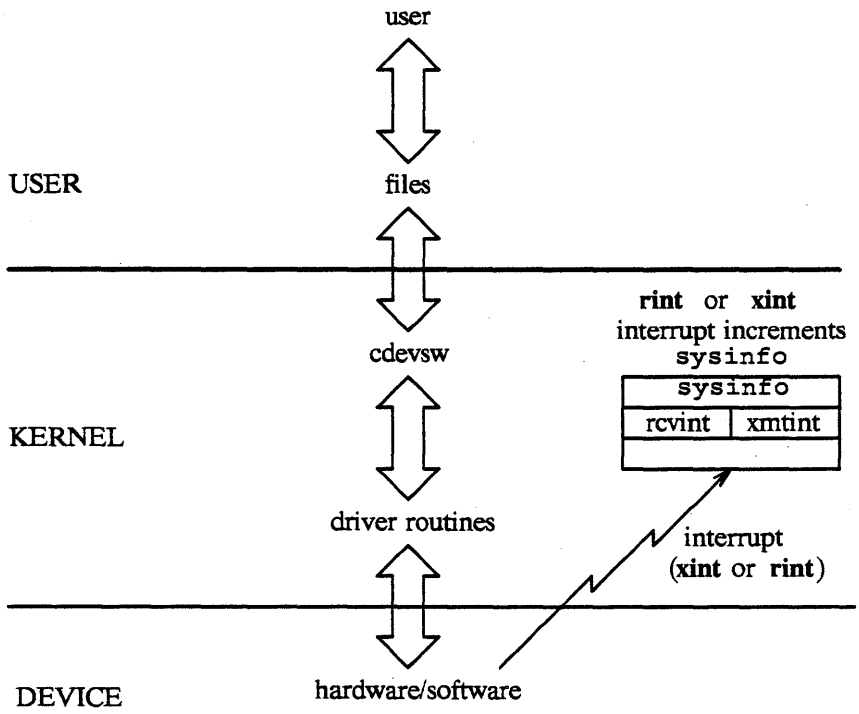


Figure D4X-16 sysinfo Structure

System activity can be accessed at the special request of a user. The **sadc**(1M) and **sar**(1M) commands also can access the the `sysinfo` structure. For further information on these maintenance commands, see the *Administrator's Reference Manual.*

A user programming example using the `sysinfo` structure is shown in Figure D4X-17.

```
1   unixfile = ldopen("/unix",NULL);   /* open /unix */
2   ldtbseek(unixfile);                /* seek to start of symb tab */
3   index = ldtbindex(unixfile);       /* go to 1st symbol */
4   while (1) {
5      ldtbread(unixfile,index,symbolstruct);      /* read tab entry */
6      if (!strcmp(symbolstruct.name,"sysinfo"))
7              break;                              /* found it */
8      index = ldtbindex(unixfile);                /* go to next symbol */
9      }
10  memfile = open("/dev/mem",O_RDONLY);/* open /dev/mem */
11  lseek(memfile,symbolstruct.address);           /* go to address of sysinfo */
12  read(memfile,&sysinfostruct,sizeof(sysinfostruct);     /* read sysinfo */
```

Figure D4X−17 sysinfo Code Example

The example program performs the following functions:

- Opens the */unix* file (line 1) and reads its symbol table (done via the **ld*** library functions (lines 2, 3, and 5); see Section 3X of the *Programmer's Reference Manual*).
- Searches the symbol table for the address of the `sysinfo` structure (line 6).
- Opens */dev/mem* (the file that is a window into memory) (line 10). Root permission is required to read */dev/mem*.
- Uses the address found in the symbol table to copy the `sysinfo` structure out of memory into the current program (line 12).

STRUCTURE MEMBERS

Type	Member	Description
long	rcvint;	/* Increment on entry to rint */
long	xmtint;	/* Increment on entry to xint */

SOURCE FILE

sysinfo.h

SEE ALSO

rint(D2X), **xint**(D2X)

tty (D 4 X)

DESCRIPTION

Character queues and buffers for a TTY driver are associated with a given TTY device through the `tty` (terminal) structure. The `tty` structure maintains all information relevant to the TTY device.

The TTY subsystem is a series of buffers in which data is manipulated. The subsystem is designed to convert raw terminal data into data usable by a user program (see Figure D4X-18).

To make the data usable, the TTY functions handle occurrences of the user pressing BREAK or DELETE , BACKSPACE , or other special characters. By pressing a keyboard key, 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
- echos characters to the **t_outq** output buffer
- resolves BREAK and DELETE key entries, signaling processes if necessary

The **ttread**(D3X) function is called to convey the data from **t_canq** to the user process.

The **ttwrite**(D3X) routine conveys the data from the user program to the **t_outq** output buffer.

The **ttout**(D3X) routine is called to convey the data from the **t_outq** output buffer to the **t_tbuf** transmit buffer.

Finally, a driver device dependent output routine sends the data to the terminal screen.

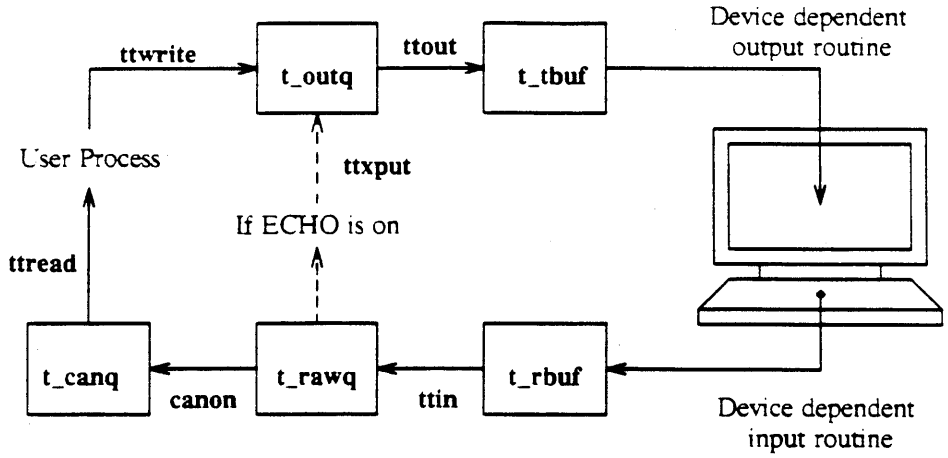


Figure D4X–18 tty Structure

STRUCTURE MEMBERS

Type	Member	Description
struct clist	t_rawq;	/* Device raw input queue head */
struct clist	t_canq;	/* Device canonical queue head */
struct clist	t_outq;	/* Device output queue */
struct cblock	t_tbuf;	/* Device transmit buffer */
struct cblock	t_rbuf;	/* Device receive buffer */
int	t_proc;	/* proc routine address */
ushort	t_iflag;	/* Input mode */
ushort	t_oflag;	/* Output mode */
ushort	t_cflag;	/* Control mode */
ushort	t_lflag;	/* Local mode */
short	t_state;	/* Device and driver internal state */
short	t_pgrp;	
char	t_line;	/* Line discipline type */
char	t_delct;	/* Number of delimiters */
unsigned char	t_cc[NCC];	/* Control characters */

The following elements of the `tty` structure are significant:

- **t_rawq** points to the first `cblock` of the device's raw input queue (before character processing is performed), a `clist`(D4X) structure
- **t_canq** points to the first `cblock` of the device's canonical queue (after character processing is performed), a `clist` structure

- **t_outq** points to the first `cblock` of the device's output queue, a `clist` structure
- **t_tbuf** is the device's transmit buffer
- **t_rbuf** is the device's receive buffer
- **t_proc** holds the address of a **proc**(D2X) driver routine. Each device driver for a TTY device must provide a special hardware-specific access or **proc** routine.
- **modes** are four members of the `tty` structure that specify the **ioctl** flags listed in **termio**(7) modes. The **t_iflag** element holds the input modes specified in the **c_iflag** element of the `termio` structure. The **t_oflag, t_cflag,** and **t_lflag** elements hold output modes, control modes, and local modes as specified in the **c_oflag, c_cflag,** and **c_lflag** elements of the `termio` structure. The contents of these fields are defined on the **termio**(7) manual page.
- **t_state** maintains the internal state of the device and the driver. Each of the 16 bits of this member is assigned to one of the items in the following list. Thus, the state is a composite of one or more of the items below. Note that the **t_state** member is fully utilized and cannot be extended for additional state information that a particular driver may need. The states are as follows:

BUSY	indicates output is in progress
CARR_ON	software image of the carrier-present signal
CLESC	indicates the last character processed was an escape character
EXTPROC	indicates a peripheral device is performing semantic processing of data
IASLP	indicates the processes associated with the device should be awakened when input completes
ISOPEN	indicates the device is open
OASLP	indicates the processes associated with the device should be awakened when output completes
RTO	indicates a timeout is in progress for a device operating in raw mode; that is, where no canonical processing is taking place
TACT	indicates a timeout is in progress for the device
TBLOCK	indicates the driver has sent a control character to the terminal to block transmission from the terminal
TIMEOUT	indicates a delay timeout is in progress
TTIOW	indicates the process associated with the device is sleeping, awaiting the completion of output to the terminal

TTSTOP	indicates output has been stopped by a CTRL-s character (ASCII DC3) received from the terminal
TTXOFF	indicates the Central Processing Unit (CPU) has hit the high water mark in receiving data from a TTY device. You now want the terminal to send a CTRL-s character to stop output. Calls the driver **proc** routine with T_BLOCK as the *cmd* argument.
TTXON	indicates the data processed by the CPU has hit the low water mark. Therefore, a CTRL-q character should be sent when the transmitter is ready. Calls the driver **proc** routine with T_UNBLOCK as the *cmd* argument.
WOPEN	indicates the driver is waiting for an open to complete

- **t_pgrp** identifies the process group associated with the device. It is needed to send signals to the process group.
- **t_line** holds the line discipline type specified in the **c_line** element of the `termio` structure
- **t_delct** used by the TTY subsystem to keep track of the number of delimiters found while performing semantic processing of data
- **t_cc[NCC]** an array holding the control characters specified in the **c_cc** member of `termio`

A character device driver using the TTY subsystem must declare an instance of the `tty` structure for each subdevice under its control.

SOURCE FILE

tty.h

SEE ALSO

BCI Driver Development Guide, Chapter 7, "Drivers in the TTY Subsystem."
`linesw`(D4X)

DESCRIPTION

Each process is allocated a `user` data structure (also known as a user block) to contain information defining the user process and its state to the kernel.

The `user` structure must never be accessed from a driver interrupt level as there is no certainty which user block is accessed. When a random interrupt occurs, there can be no assurance the user block you manipulate is the one associated with the interrupt event. In addition, in some computers such as the 3B2 computer, the user block may be swapped out.

The `user` structure contains information that is needed only when the process is running. The **u.u_base** member specifies the virtual address for I/O to and from the user data area. Information is transferred from the individual user block to the kernel user structure (see Figure D4X-19).

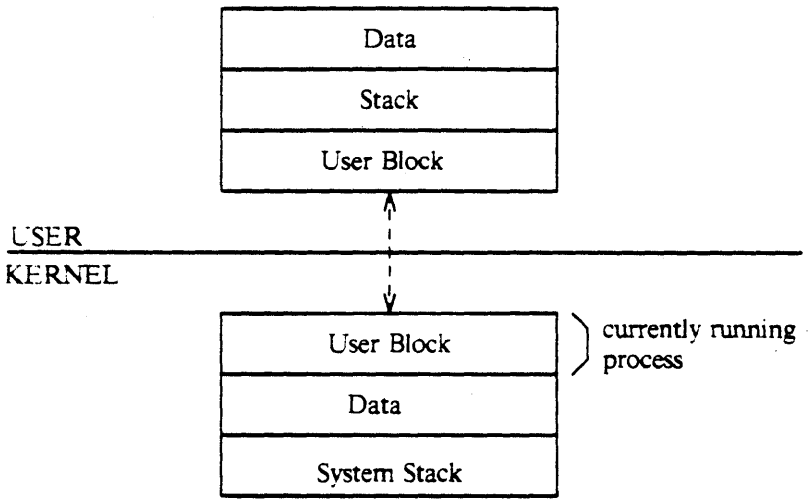


Figure D4X-19 user Structure

The `user` structure for the current process is always a fixed address in the operating system address space. The kernel can look for the `user` structure only for the currently running process. Since the `user` structure is basic to the kernel, it is subject to change from one software release to another. Since some drivers require access to certain fields of this data structure, the following fields will not be subject to change.

STRUCTURE MEMBERS

Type	Member	Description
caddr_t	u_base;	/* I/O base address */
unsigned	u_count;	/* Bytes remaining for I/O */
char	u_error;	/* Return error code */
ushort	u_gid;	/* Effective group ID */
off_t	u_offset;	/* Offset into file for I/O */
struct proc	u_procp;	/* proc structure pointer */
label_t	u_qsav;	/* Quits and interrupts variable */
ushort	u_rgid;	/* Real group ID */
ushort	u_ruid;	/* Real user ID */
char	u_segflg;	/* User or kernel I/O flag */
short	u_ttyp;	/* 3B2 tty structure pgrp pointer */
ushort	u_ttyp;	/* 3B4000 tty structure pgrp pointer */
ushort	u_uid;	/* Effective user ID */

In addition to the above structure members, the **u.u_r** union is used to return values to system calls.

The members of the `user` structure are described as follows:

u_base specifies the virtual base address for I/O to and from user data space.

u_count specifies the number of bytes not yet transferred during an I/O transaction.

u_error returns an error code (see *errno.h*) to the kernel which is then passed on to the user. This is set by a driver to indicate an error condition. See **intro**(2) in the *Programmer's Reference Manual* for a description of available error codes for setting error codes. Also see **copyin**(D3X) for an example of the **u.u_error** member.

u_offset specifies the offset into the file from which or to which data is being transferred.

u_procp address of the `proc` structure associated with this `user` structure.

u_r returns values to system calls.

u_qsav	field used as an argument to the kernel **longjmp**(D3X) function. This field is set up automatically before a driver is called. Therefore, a driver can use **longjmp** with **u.u_qsav** to stop normal processing when an error is detected in the base level. The base level consists of the routines in a driver that interact with the kernel.
u_ruid and **u_rgid**	identifies the real user and group IDs.
u_segflg	determines what type of I/O transfer is to occur. The driver should set this field to 1 to indicate data movement within the kernel space; set it to (zero) to indicate data movement between kernel space and user space. Always save the previous value of **u.u_segflg** before changing it and restore the previous value when you have completed your I/O transfer.
u_ttyp	address of the the `tty` structure for the controlling terminal.
u_uid and **u_gid**	processes effective user and group identification members. **u.u_uid** and **u.u_gid** may be used to provide a process identified by the user and group identification members (**u.u_ruid** and **u.u_rgid**) with the access permissions of another process or process group.

Table D4X-2 lists `user` structure members that do not vary between UNIX System releases and that can be set or read.

Table D4X–2 user Structure Member Uses

Member	Use
u_base	driver setable
u_count	driver setable
u_error	driver setable
u_gid	read only
u_offset	driver setable
u_procp	read only
u_qsav	read only
u_rgid	read only
u_ruid	read only
u_segflg	driver setable
u_ttyp	driver setable
u_uid	read only

SOURCE FILE

user.h

SEE ALSO

BCI Driver Development Guide, Chapter 4, "Header Files and Data Structures"

Contents

Introduction

Section D8X lists a subset of the standard library macros used to write or maintain a diagnostics file for a 3B2 computer or 3B4000 ACP feature card (circuit board). The subset provided is the minimum number of macros required to write or maintain diagnostics files. Information on other macros will be provided in subsequent releases of this manual. In addition, because little information is known about the macros described in this section, most of the headings provided in Section D3X are not provided. This section augments the information provided the *BCI Driver Development Guide* in Appendix B, "Writing 3B2 Computer Diagnostics Files".

A diagnostic file passes information to an intelligent controller so that the system initialization software can ensure the integrity of a 3B2 computer or 3B4000 ACP feature card (circuit board). Each hardware driver requires two diagnostics files and these files are stored in the *dgn* directory. Both file names are in upper case 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 feature card object code for the diagnostic tests.

Table D8X–1 Function Summary

Function	Description
CLEANUP()	initialize board registers
EDTP(*element*)	return pointer to *element* of the EDT
EXCRET()	set up return point for exception
GETS(*ptr*)	get string from standard input
GETSTAT()	return value of current console character
LONGJMP(*array*)	return to point set by SETJMP
NUM_EDT()	return number of entries in EDT
PRINTF("*string %options*",*arg1*,*arg2*)	display message
SETJMP(*array*)	set sane return point
SSCANF(*string*,"*%options*",*arg1*,*arg2*)	read from *string*
STRCMP(*string1*,*string2*)	compare strings

CLEANUP(D8X)

SYNOPSIS

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

        CLEANUP();
```

DESCRIPTION

CLEANUP initializes the control status register (CSR), the direct memory access (DMA) controller, the sanity and interval time, and the floppy disk interrupts.

E D T P (D 8 X)

SYNOPSIS

```
#include <sys/edt.h>
#include <sys/firmware.h>

        int
        EDTP(element);
```

DESCRIPTION

EDTP returns a pointer to the *element* of the equipped device table (EDT). The *element* argument can be any number between 0 and NUM_EDT-1.

EDTP is used with NUM_EDT(D8X) to get the maximum number of entries in the EDT. The return value from EDTP should be assigned to the **edt_pointer** member of the `edt` structure (defined in *edt.h*). The following example returns a pointer to the last device in the EDT.

```
#include <sys/edt.h>
#include <firmware.h>

        struct edt *edt_pointer;
        int N;
        N = NUM_EDT;        /* Set N to maximum number of devices in EDT */

        edt_pointer = EDTP(N - 1);  /* point edt_pointer at last EDT entry */
```

Figure D8X−1 EDTP(D8X) Example

EXCRET(D8X)

SYNOPSIS

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

        char
        EXCRET();

        extern myexec_handler();
        extern myint_handler();

        EXC_HAND=myexec_handler;
        INT_HAND=myint_handler;
```

DESCRIPTION

In diagnostics programming, you may wish to force an exception on a board to ensure that the hardware can actually handle it as expected. The EXCRET macro sets up a return point to return to should an exception occur. Using EXCRET requires one of the following methods:

- Execute the exception and the call to EXCRET in the same routine.
- Use SETJMP(D8X) and LONGJMP(D8X); immediately after the call to the exception routine, test the return flag (the flag must be declared **extern**). If an exception occurred (indicated by the return flag), call LONGJMP to return to the point set by SETJMP.

EXC_HAND and INT_HAND control how the exception is handled. These are explained as follows:

EXC_HAND	if an exception occurs, control is passed to the routine set in EXC_HAND. When the exception handler returns, control passes to the instruction immediately after the last call to EXCRET.
INT_HAND	if an interrupt occurs, the routine specified in INT_HAND is called and executed. A return from the process is executed after a normal return by the exception handler. This lets program flow continue where it left off before the interrupt was serviced.

Default exception and interrupt handlers print an error message and fail the diagnostic. EXCRET should be called prior to any exception and within the same routine as the expected exception.

Two locations are provided for storing addresses of interrupt and exception handlers for diagnostics. These two locations are referenced using

```
1   phase()
2   {
3   extern my_int_handler();
4   extern my_exc_handler();
                    .
                    .
                    .
5   INT_HAND=my_int_handler;
6   EXC_HAND=my_exc_handler;
                    .
                    .
                    .
7   EXCRET;
                    .
                    .
                    .
8   }
```

Figure D8X-2 Using EXCRET(D8X)

GETS(D8X)

SYNOPSIS

#include <sys/firmware.h>

short
GETS(*buf*);
char **buf*;

DESCRIPTION

GETS returns a NULL terminated string from standard input with the carriage return stripped out. The string is placed in *buf*. GETS is disabled when the INPUT/OUTPUT flag is OFF.

Input to GETS is limited to 79 characters (GETS_SIZE -1 for the NULL character). GETS_SIZE is defined in *firmware.h* and is typically set to 80. Input that exceeds this limit results in the following warning message being displayed:

```
max input of 80 characters, re-enter line
```

GETSTAT(D8X)

SYNOPSIS

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

        char
        GETSTAT();
```

DESCRIPTION

GETSTAT returns the value of the current character in the console receiver if one is present; otherwise, a zero is returned. The returned value allows a phase to build its own console driver indirectly or to implement a kill character, such as break, by periodic polling.

GETSTAT is used as follows:

```
c=GETSTAT();
```

LONGJMP(D8X)

SYNOPSIS

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

        long
        LONGJMP(array);
```

DESCRIPTION

LONGJMP returns control to the point at which a previous SETJMP(D8X) call was made. *array* is a 12-element array of integers. A one-to-one correspondence exists with the *array* name between the LONGJMP and the SETJMP calls.

N U M _E D T (D 8 X)

SYNOPSIS

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

        int
        NUM_EDT();
```

DESCRIPTION

NUM_EDT returns the number of entries in the equipped device table (EDT).

PRINTF(D8X)

SYNOPSIS

#include <sys/firmware.h>

```
short
PRINTF("string %options",arg1,arg2);
```

DESCRIPTION

PRINTF is subset of **printf**(3S). PRINTF supports the **%-, %O, %c, %S, %d, %o, %x,** and **%u** specifications.

SETJMP(D8X)

SYNOPSIS

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

int array [12];

        long
        SETJMP(array);
```

DESCRIPTION

SETJMP sets a point to which LONGJMP(D8X) can return control. SETJMP saves all registers. If SETJMP returns true, then a LONGJMP has occurred. If SETJMP returns false, then this is the first call to SETJMP.

When first executed, SETJMP saves the state of the stack and register variables in *array* and returns 0. After setup, any call to LONGJMP transfers control to program counter that was stored by the SETJMP call. After LONGJMP is executed, SETJMP returns 1. Any number of jump buffers may be used for independent LONGJMP calls. Using 0 (NULL) for the jump buffer selects a global default jump buffer. This macro has different arguments than does the **setjmp**(3C) system call.

SSCANF(D8X)

SYNOPSIS

#include <sys/firmware.h>

```
long
SSCANF(string, "%options",arg1, ... argn);
char *string;
short *arg1, ... *argn;
```

DESCRIPTION

SSCANF is a modified version of **sscanf**(3S). The options recognized are **%s** for strings, **%c** for chars, **%D** for long decimal, **%d** for short decimal, **%X** for long hex, and **%x** for short hex. Delimiters in input are space, tabs, commas, or dashes.

NOTE: Ensure that the type of the argument matches the type of the option. For example

```
short short_parm;
long long_parm;
SSCANF(string, "%d %D", &short_parm, &long_parm);
```

STRCMP(D8X)

SYNOPSIS

#include <sys/firmware.h>

char
STRCMP(*string1*, *string2*);

DESCRIPTION

STRCMP is a subset of **strcmp**(3F). Zero is returned for equal, -1 if *string1* is less, and 1 if *string1* is greater.

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*

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 Reference Manual, Issue 2 **307-192**

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