Kernel Reference Manual

AEG

REAL/IX® Operating System

Open Architecture Systems

211–871001–000

MODCOMP

AEG

Kernel Reference Manual

REAL/IX® Operating System
Open Architecture Systems

MODCOMP

MODULAR COMPUTER SYSTEMS, INC.
1650 WEST McNAB ROAD
P.O. BOX 6099
FORT LAUDERDALE, FL 33340-6099

THIS MANUAL IS SUPPLIED WITHOUT REPRESENTATION OR WARRANTY OF ANY KIND. MODULAR COMPUTER SYSTEMS, INC. THEREFORE ASSUMES NO RESPONSIBILITY AND SHALL HAVE NO LIABILITY OF ANY KIND ARISING FROM THE SUPPLY OR USE OF THIS PUBLICATION OR ANY MATERIAL CONTAINED HEREIN.

CLASSIC, CLASSIC REAL/IX, MAX, MODCOMP, MODMate, GLS, PACE/32, PACE/IX, PRIDE, REAL/IX, REAL/STAR, REAL/STONE, REAL/VU, Tri-Dimensional, Tri-D, TRI-SIM, and Modular Computer Systems are registered trademarks of Modular Computer Systems, Inc.
Development Server, PROTOP/IX, and REAL/MAX are trademarks of Modular Computer Systems, Inc.

The following are trademarks or **registered** trademarks of their respective companies: Macintosh of Apple Computer, Inc.; **Documenter's Workbench, TELETYPE** of AT&T; VCL, EDT+ of Boston Business Computing, Ltd.; Belden of Cooper Industries, Inc.; Centronics of Data Computer Corporation; DEC, DECnet, PDP, VAX, VMS of Digital Equipment Corporation; **EXOS, Excelan** of Excelan, Inc.; HP, Vectra of Hewlett-Packard Company; VRTX/68000 of Hunter & Ready, Inc.; BITBUS, **MULTIBUS**, Intel-(followed by any number) of Intel Corporation; **IBM**, PC-DOS, **PC/AT, PC/XT, PS/2** of International Business Machines Corporation; X Window System of Massachusetts Institute of Technology; **Microsoft, Windows, MS-DOS, Xenix** of Microsoft Corporation; MODICON, Modbus, Modbus Plus of Modicon, Inc.; **Motorola**, System V/68, VME Delta Series, Delta Series, VMEexec, Business Assistant, OfficeLAN, VERSAbus, VERSAdos, 133Bug, 147BUG, 187Bug, HYPERmodule, VMEmodule, 188Bug, SSID of Motorola, Inc.; OSF, OSF/Motif, Motif of Open Software Foundation, Inc.; **ORACLE, SQL*Calc, SQL*Forms, SQL*Plus**, SQL*Graph, **SQL*Menu, SQL*Net**, Pro*SQL, Pro*ORACLE, Pro*C, Pro*COBOL, Easy*SQL of Oracle Corporation; Q-ONE of Quadretron Systems Inc.; FORTRIX of Rapitech Systems, Inc.; **PROBUG** of SBE, Inc.; **Sun**, NFS, ONC of Sun Microsystems, Inc.; **TEKTRONIX** of TEKTRONIX, Inc.; **TeleSoft**, Telegen2 of **TeleSoft; TeleVideo** of TeleVideo Systems, Inc.; **UNIX** of UNIX System Laboratories, Inc.; **DIABLO, Ethernet, Xerox** of Xerox Corporation; **88open**, BCS, OCS of 88open Consortium, Ltd..

Contents subject to change without notice.

Portions of this document are based on or reprinted from copyrighted documents by permission of Motorola, Inc. and AT&T.

Service and Assistance

MODCOMP offers a variety of programs and services that demonstrate our commitment to customer satisfaction. Our Technical Education department provides comprehensive hands-on instruction either at our facilities or at customer-designated sites. Our worldwide field service organization is ready to give installation assistance, free service during the warranty period, and flexible service programs tailored to your requirements.

Your MODCOMP sales and service representatives can help you with any questions, problems, or suggestions you may have for our products and services. For your convenience, MODCOMP maintains toll-free telephone numbers at which we can be reached. Numbers you may find helpful are listed here.

For	Call	From
Questions, sales information, or suggestions	• 1–800–255–2066	• U.S.A. and Canada
	• 1–305–974–1380 extension 1800	• outside the U.S.A. and Canada
	or please call your regional support office	
Service	• 1–800–327–8928	• U.S.A.
	• 1–416–890–0666	• Canada
	Outside the U.S.A. and Canada, please call your regional service/support office.	
Technical Education information	• 1–305–977–1708	• U.S.A.
	Outside the U.S.A., please call your regional support office.	

Contents

Page

Page

Chapter 3 Kernel Functions and Macros (D3X) [continued]

Page

Chapter 3 Kernel Functions and Macros (D3X) [continued]

Page

Chapter 3 Kernel Functions and Macros (D3X) [continued]

Page

Chapter 3 Kernel Functions and Macros (D3X) [continued]

Chapter 4 Data Structures (D4X)

Page

Chapter 4 Data Structures (D4X) [continued]

Tables

Manual History 211−871001

REAL/IX Operating System, Kernel Reference Manual

This section identifies each issue of this manual and lists them in reverse chronological order. Except for the initial issue, a detailed summary explains the changes made in each of the five most recent revisions.

Revision 000 (Initial Issue) 02/94

About This Manual

The *Kernel Reference Manual* provides reference material about the driver entry-point routines, kernel functions, and kernel data structures used to write device drivers and system calls for the REAL/IX Operating System. This manual should be used in conjunction with other books in the documentation set, especially the *Kernel Programming Guide* and the *Driver Development Guide*.

Open Architecture Systems Defined

The term "open architecture system", in its simplest form, implies that a user may add a variety of vendors' components to a single system. This is possible when certain industry-accepted standards have been implemented in the system. MODCOMP open architecture systems are based on such software and hardware standards as the UNIX System V operating system; VMEbus, MULTIBUS, and SCSI bus interfaces; and CPUs built around standard microprocessors. By building on these standards, open architecture systems provide computer solutions that are portable and compatible.

The REAL/IX Operating System[1] allows applications to be ported easily between traditional UNIX systems and MODCOMP open architecture systems. Furthermore, by using industry-standard I/O buses, MODCOMP open architecture systems ensure compatibility among a wide range of peripheral and I/O devices and the ability to expand as needs dictate. MODCOMP open architecture systems meet networking and communications needs with such industry standards as Ethernet and TCP/IP and have the flexibility to accommodate new standards as they are developed.

[1] The REAL/IX Operating System, featuring realtime and multiprocessing capabilities, is the MODCOMP implementation of the UNIX System Laboratories UNIX System V operating system.

Identifying Platform–Specific Information

The information in this manual applies to the REAL/IX Operating System running on various hardware platforms. Although most of the information applies to all REAL/IX systems, some functions and operations are platform specific. These areas are identified by the icons shown below.

386 — Identifies information that applies only to computers with a 386 or higher microprocessor (e.g., 486), regardless of the I/O bus

ISA — Identifies information that applies only to computers with a 386 or higher microprocessor (e.g., 486) and the Industry Standard Architecture (ISA) I/O bus

MBII — Identifies information that applies only to computers with a 386 or higher microprocessor (e.g., 486) and the Multibus II I/O bus

MP — Identifies information that applies only to computers running in a multiprocessing environment

VMEbus — Identifies information that applies only to VMEbus-based computers, regardless of the microprocessor

VMEbus 88K — Identifies information that applies only to VMEbus-based computers with the M88000 Series RISC microprocessor

Related Publications

This section lists suggested sources of additional information. Publications are listed under appropriate categories; platform-specific publications are grouped together within each category. Contact your Sales Representative to order.

Books for All System Users

Concepts and Characteristics
Gives an overview of the internals of the REAL/IX Operating System and an introduction to the tools and facilities that are available.

User's Guide
Discusses basic user procedures including the login procedure and getting around the file system. Information is included about general user tools; for example, the **vi** and **ed** text editors, electronic mail, the shell programming language, and the Korn shell.

Using UUCP and Usenet
Introduces UUCP communications, describes how to transfer files and execute remote commands over UUCP, how to check on UUCP requests, and how to access the Usenet electronic bulletin board.

VMEbus 88K

POSIX Conformance Guide
Describes conformance to IEEE Std 1003.1–1988. This document describes only those areas where the specification allows implementation-defined behavior, or where the behavior of an implementation may vary.

VMEbus 88K

Reference Manuals
Multi-volume set of manual pages describing user, administrative, and real-time commands (Sections 1, 1M, and 1R); system calls, library routines, and miscellaneous facilities (Sections 2, 3, and 5); and system files, special device files for standard devices, and special device files for add-on packages (Sections 4, 7, and 7A).

ISA

Reference Manuals
Multi-volume set of manual pages describing user, administrative, and real-time commands (Sections 1, 1M, and 1R); system calls, library routines, and miscellaneous facilities (Sections 2, 3, and 5); and system files, special device files for standard devices, and special device files for add-on packages (Sections 4, 7, and 7A).

MBII

Intel System V/386 MULTIBUS Reference Manual (Intel Order Number: 463328–001)
Contains manual pages for user commands, file formats, device drivers, and maintenance commands.

Books for System Administrators

Software Installation Guide

Gives instructions for installing the operating system (either for the first time or as an upgrade) and initially setting up the system.

System Administrator's Guide

Gives instructions and background information about administering the REAL/IX Operating System. Topics covered include ensuring system security; creating and maintaining user and group IDs; working with file systems (creating, repairing, backing up); setting up terminals and printers; using the **sysgen**(1M) utility to modify tunable parameters and to configure or deconfigure standard system devices; and setting up and using the Job Accounting System. Appendixes discuss the system files that control system operations and the file naming conventions for special device files.

Managing UUCP and Usenet

Provides background information about UUCP for administrators and gives instructions for setting up a UUCP link, verifying that the link works, administering UUCP communications, and setting up and administering the Usenet access. This information is supplemented by the *System Administrator's Guide*, which includes information for administering UUCP over the TCP/IP protocol, and the *Software Installation Guide*.

VMEbus

Software Engineering Release Notes

Gives an overview of the new features in this release of the REAL/IX Operating System and provides usage notes for the system.

VMEbus

System Guide

Gives an overview of MODCOMP open architecture systems VMEbus-based computers and contains instructions for the installation and maintenance of these systems.

VMEbus
88K

Guide to VME Modules, MVME187 Host CPU
Guide to VME Modules, MVME197 Host CPU
Guide to VME Modules, MVME188 RISC Board Set

Provide installation and hardware setup information, and firmware-level initialization information for VMEbus-based systems.

386

Software Engineering Release Notes

Gives an overview of the new features in this release of the REAL/IX Operating System and provides usage notes for the system.

Books for Programmers

Languages and Support Tools Guide
Provides tutorials for many of the special purpose languages and the programming support tools available on the REAL/IX Operating System.

Programmer's Guide
Gives an overview of the REAL/IX Operating System and realtime computing, describes the REAL/IX programming environment and the operating system interface, and provides programming examples for using the realtime extensions of the REAL/IX Operating System as well as the standard UNIX operating system features.

The C Programming Language, Second Edition
Describes the ANSI C language.

386

AT&T *UNIX System V/386 Release 3.2 Programmer's Guide* (1988)
Focuses on programming elements that are part of getting programs into operation in the UNIX System V/386 operating system environment.

MBII

Intel System V/386 MULTIBUS II Transport–Application Interface Guide
(Intel Order Number: 463116-001)
Describes the application interface to MULTIBUS II transport using MULTIBUS II message passing.

Books for Kernel Programmers

Driver Development Guide
Introduces the process of writing device drivers for the REAL/IX Operating System, including detailed information about porting and installing drivers.

Kernel Programming Guide
Gives background information about topics of interest to programmers writing device drivers and system calls. Topics discussed include how drivers and system calls execute and how various types of I/O operations are implemented.

Kernel Reference Manual
Contains reference pages for driver entry-point routines (Section D2X), kernel functions and macros (Section D3X), and kernel data structures (Section D4X) used for coding system calls and device drivers.

MBII

Intel System V/386 MULTIBUS II Device Driver Guide
(Intel Order Number: 463463–001)
Provides the information needed to write MULTIBUS II transport calls and interconnect-space calls within a device driver. Also defines Intel's static bad block handling, as well as disk-specific information unique to Intel Systems.

Industry Standard Publications

The REAL/IX Operating System and its supported C programming language comply with the industry standards listed below. These standards are commercially available and can be obtained from the following sources. While an effort was made to ensure that the ordering information was complete and up-to-date at time of printing, we cannot guarantee its accuracy.

ANSI X3.159–1989 *Programming Language C Standard*
American National Standards Institute, Inc.
Sales Department
1430 Broadway
New York, NY 10018
Phone: (212) 642–4900
Fax: (212) 302–1286

System V Interface Definition (SVID)
AT&T Customer Information Center (CIC)
Customer Service Representative
P.O. Box 19901
Indianapolis, IN 46219
Phone: 1–800–432–6600 (Inside U.S.A.)
1–800–255–1242 (Inside Canada)
(317) 352–8557 (Outside U.S.A. and Canada)

VMEbus 88K

IEEE Std 1003.1–1988
Standard Portable Operating System Interface for Computer Environments (POSIX)
The Institute of Electrical and Electronics Engineers, Inc.
Publications Sales, IEEE Service Center
P.O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855–1331
Phone: 1–800–678–IEEE (4333)
FAX: (201) 981–9677

VMEbus 88K

88open Binary Compatibility (BCS)
88open Consortium Ltd.
Marketing Department
100 Homeland Court, Suite 800
San Jose, CA 95112
Phone: (408) 436–6600
Fax: (408) 436–0725

Documentation Conventions

The following table gives the textual conventions used in this book. Note that commands, library routines, system calls, kernel functions, driver entry points, files, and data structures are sometimes followed by a number enclosed in parentheses (for instance, "**cat**(1)"). This denotes the reference section in which they are located; Sections D2X, D3X, and D4X are in the *Kernel Reference Manual*; all others are in the *Reference Manual* volumes and available online through the **man**(1) command. Commands followed by empty parentheses (for instance, "**false**( )") are available through the **man** command, but do not have their own manual page.

Style	Item	Example
bold	Shell commands	**cat** *or* **cat**(1)
bold	Library routines	**printf** *or* **printf**(3s)
bold	System call names	**open** *or* **open**(2)
bold	Kernel function names	**copyin** *or* **copyin**(D3X)
bold	Driver entry point names	**strategy** *or* **strategy**(D2X)
bold	Script names	**MOUNTFSYS** *or* **S03MOUNTFSYS**
italics	File names	*/etc/passwd*
`monofont`	Data structures	`user` *or* `user`(D4X)
bold	Data structure members	**u_count** *or* **u.u_count**
bold	Literal text in example	**cat** *filename*
italics	Variable text in example	
`monofont`	Code representations	`if size <= 0 return NULL;`
`monofont`	Screen representations	`Enter a number or q to quit: 2`
`monobold`	Operator input	
?	Single character wildcard	*/dev/tty??*
*	Multi–character wildcard	/dev/*_ct
⚠ WARNING!	Highlights information that, if not observed, could cause bodily harm.	

CAUTION	Highlights information that, if not observed, could cause the system or a procedure or practice to fail or could damage existing data on the system.
NOTE	Highlights relevant information that does not require a caution or warning.
HINT	Identifies material that is indirectly related to the subject matter being discussed. For instance, a procedure may specify one way of doing the task, and the HINT explains why it is done this way or suggests optional ways to accomplish the same task.

Chapter 1

Introduction

The *Kernel Reference Manual* for the REAL/IX® Operating System provides information needed by programmers who wish to add system calls and device drivers to the REAL/IX Operating System. It is based on the AT&T *Block and Character Interface (BCI) Driver Reference Manual.*

Note that the programming code samples in the *Kernel Reference Manual* are code fragments that are intended to demonstrate the use of the entry point, function, data structure, or library function being described. These code fragments are not intended to be compiled into drivers.

The kernel programming documentation for the REAL/IX Operating System defines the terms routine and kernel function as follows:

routine — Code segment written by a driver developer. Driver code consists of entry-point routines and subordinate routines. The entry-point routines are accessed through system tables and must be named according to very specific rules that are explained in the introduction to Section 2 of this book. Subordinate driver routines are called by driver entry-point routines.

function — A kernel utility used in a driver or system call. The use of functions in kernel-level code is analogous to the use of system calls and library routines in user-level code.

Organization of This Book

This book uses the AT&T format, a format similar to that used in the standard UNIX® reference manuals. After this introduction, the book contains three sections:

D2X contains manual pages for the entry-point routines that form the skeleton of any driver code. Each page discusses what the entry-point routine does, identifies any configuration dependencies associated with the routine, and gives guidelines for writing the routine. A table in the introduction compares the supported entry-point routines to those documented by AT&T.

D3X contains manual pages for the kernel functions that are used instead of library functions in device drivers and system calls. Each page gives a synopsis of the function (including any header files that must be called when using it), describes the return codes for the function, specifies any semaphoring ramifications, tells whether the routine can be used from base or interrupt level, and identifies the file in which the source for the function is located (customers with binary licenses may not have all the source files referenced). Tables in the introduction to the section summarize all documented kernel functions and compare the function set to that documented by AT&T.

D4X contains manual pages for the kernel data structures that may be accessed by drivers and system calls. Each page describes the use of the structure, defines the structure members that may be accessed, and identifies the file in which the structure is defined (in most cases, the structure is defined in a header file located in the */usr/include/sys* directory; these files are included in the binary release).

This book should be used in conjunction with two other books in the documentation set for the REAL/IX Operating System:

- ❑ *Kernel Programming Guide* provides background information covering a number of topics involved in writing device drivers and system calls.
- ❑ *Driver Development Guide* introduces the specific tasks involved in writing and porting device drivers for the REAL/IX Operating System.

Refer to the Preface of this book for a list of other books in the documentation set.

Porting Driver Code

When discussing the portability of kernel-level code, it is important to remember that there is no standard on kernel code: neither SVID nor POSIX addresses anything below the system-call level, and all that is standardized for system calls is a basic set to be included, not the lower-level kernel functions used to implement system calls. Consequently, each kernel has a number of variations from other kernels. In addition to modifications made to provide performance that is acceptable for realtime applications, the REAL/IX kernel includes some modifications to the UNIX System V kernel made when the operating system was ported to the hardware platform on which your machine is based.

As a starting point, the tables at the beginning of Sections 2 and 3 compare the REAL/IX kernel to that documented in the AT&T UNIX System V Release 3 *Driver Reference Manual*. If the kernel code you are porting ran on a different variation of the operating system, you may find additional inconsistencies. At worst, these changes could be a minor aggravation. If you have code to port, a simple **grep**(1) should enable you to identify all UNIX System V entry-point routines and kernel functions that are not supported. To identify other variations, you can carefully compare the code to the routines and functions listed in the beginning of Sections 2 and 3, or you can attempt to compile the driver code; the linker will flag unsupported functions as unresolved references.

For more information about porting issues, refer to *Portable C and UNIX System Programming* (Lapin 1987). Lapin explains the relationships between the various UNIX dialects, points out common pitfalls when porting code, and provides some helpful insight into writing portable C code. Of particular interest is the section describing a portable interface to the version-dependent features of TTY drivers.

Compatibility Modes

The REAL/IX kernel uses kernel semaphores and spin locks to synchronize processes in the preemptive kernel. Compatibility modes are provided to enable you to port existing drivers to the REAL/IX Operating System without having to rewrite the drivers to use the REAL/IX synchronization facilities. These compatibility modes are specified to **sysgen**(1M) when you install the driver.[1]

- ❏ CPU affinity – Preemption is turned off whenever the driver is executing. Synchronization is done using **spl***(D3X) and **sleep**(D3X)/**wakeup**(D3X) functions, just as on UNIX System V.

- ❏ major device semaphoring – a semaphore is locked for the major number itself. Synchronization is done using **sleep/wakeup** calls; **spl*** calls that protect data structures used only by this driver can be removed.

- ❏ minor device semaphoring – a semaphore is locked for each minor number (subdevice) controlled by the driver. **sleep/wakeup** calls are used for synchronization; **spl*** calls that protect data structures used only by the driver can be removed. The interrupt-handling code must be rewritten so that the **intr**(D2X) routine determines whether the interrupt can be handled and, if not, queues it up for servicing at a later time. The **serv**(D2X) routine contains the actual interrupt-handling functionality.

One driver cannot mix **sleep/wakeup** calls with kernel functions for semaphores (such as **psema/vsema**). Some D3X kernel functions have different forms if they are used in drivers installed under compatibility modes rather than being used in fully-semaphored drivers and system calls. Special ramifications for compatibility modes are discussed on each manual page.

All user-installed system calls must be written as fully semaphored.

Refer to the *Kernel Programming Guide* for a more complete discussion of synchronization facilities for fully-semaphored kernel code versus compatibility mode driver code. The *Driver Development Guide* includes instructions for installing drivers under the compatibility modes and rewriting ported drivers to be fully semaphored.

[1] Not all compatibility modes are supported on all machines. Refer to the Release Notes shipped with your system.

Chapter 2

Driver Routines (D2X)

Section D2X describes the system entry-point routines[1] a driver developer uses to create a driver, plus the **proc** routine that is required for TTY drivers. The routines are presented on separate pages. All manual 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 configuration file. This prefix is prepended to the routine name to form the name of the actual routine in the driver. For a driver with the "pre_" prefix, for example, the driver code may contain routines named **pre_open**, **pre_close**, **pre_init**, **pre_intr**, and so forth.

Driver routines can call subroutines that are assigned names by the driver writer. Subroutines can be type **static**, in which case no rules apply for naming subroutines.[2] However, using the prefix in subroutine names enhances code readability.

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

Manual pages in this section contain the following headings:

NAME summarizes the routine's purpose

SYNOPSIS describes the routine's entry point in the source code. Note that the `#include` lines listed for the routines do not include the header files that are required for every driver; refer to the *Driver Development Guide* for information about these standard header files.

ARGUMENTS describes arguments required to invoke the routine

[1] System entry-point routines are called from the switch tables (bdevsw(D4X) and cdevsw(D4X)) during system initialization when a user-level process issues a call that activates the driver, and when a device generates an interrupt.

[2] Note that **static** symbols are not stored in the symbol table and so are not accessible to debugging tools such as **crash**(1M) and **kdb**(1M).

DESCRIPTION provides general information about the routine

RETURN VALUE describes the return values and messages that may result from invoking the routine

DEPENDENCIES lists possible dependent routine conditions

SEE ALSO lists sources of additional information. The following abbreviations are used:

KPG for the *Kernel Programming Guide*
DDG for the *Driver Development Guide*

Overview of Driver Routines

Table 2–1 lists the driver routines presented in this section. Refer to individual manual pages in this section for details about each routine.

Table 2–1. Driver Routine Types

Base-Level Routines				
System Defined Name Routines			**Subordinate Driver Routines**	
Initialization Routines	**Switch Table Accessed Routines**		**Support Routines**	**proc Routine**
Form: *prefix***init( )**	Form: *prefix***name(***args***)** **name** must be:		Form: *prefix***name(***args***)** **name** is developer selected *prefix* is not needed if the routine is declared **static**; all **static** routines are local to the driver so cannot conflict with other drivers	Form: *prefix***proc(***args***)** required for TTY drivers doing canonical processing
	Character Driver	**Block Driver**		
	open **copen** **close** **cclose** **read** **write** **ioctl** **aio** **select**	**open** **bopen** **close** **bclose** **strategy** **mbstrategy** **print** **dump**		

Interrupt-Level Routines	
Interrupt Envelope Accessed Routines	**Support Routines**
Form: Block or character driver *prefix***intr(***arg***)** *prefix***serv(***arg***)**	Form: *prefix***name(***args***)** **name** is developer selected *prefix* is not needed if the routine is declared **static**

Porting Issues

Table 2-2 summarizes the differences between UNIX System V entry points and REAL/IX Operating System entry points. If you are porting from a different operating system, you may find other variations of names, especially for the initialization and interrupt-handling routines.

Table 2–2. REAL/IX Driver Entry Points

AT&T UNIX System V Release 3		REAL/IX Release C.0 and Later	
pref**start( )**	alternate initialization entry point	Not supported; use _prefix_**init( )** for all driver initialization.	
pref**open( )**	one **open** for block or character device	_pref_**open( )**	Used for devices that code same functionality for **open** as a block or character device.
		pref**copen( )**	Optional entry points to allow driver to distinguish between **open** as block or character device.
		pref**bopen( )**	
pref**close( )**	one **close** for block or character device	_pref_**close( )**	Used for devices that code same functionality for **close** as a block or character device.
		pref**cclose( )**	Optional entry points to allow driver to distinguish between **close** as block or character device. The **close** routine must match the **open** routine used (i.e., **open–close**, **bopen–bclose**, **copen–cclose**).
		pref**bclose( )**	
pref**strategy( )**	handles block I/O operations	_pref_**strategy( )**	Used as for AT&T systems.
		pref**mbstrategy( )**	Drivers for disk devices may also include this routine, to provide the multi-block clustering feature for more efficient file access.
--	--	_pref_**aio( )**	Provides asynchronous read and write operations for block and character devices.
--	--	_pref_**dump( )**	Saves kernel memory images to supported block devices.
--	--	_pref_**select( )**	Check whether a character I/O operation started at this time will block.

pref**int( )**	interrupt handler	_pref_**intr( )**	One interrupt-handling routine is supported.
pref**rint( )**	handle receive interrupt		
pref**xint( )**	handle transmit interrupt		
--	--	_pref_**serv( )**	Required with drivers that are semaphored on the minor device.

NAME

aio – initiate asynchronous I/O operation

SYNOPSIS

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

prefixaio(cmd, areq)
int cmd;
struct areq *areq;
```

ARGUMENTS

cmd an operation that the **aio** routine performs. Typically, the driver encodes a **case** statement for each command with code to perform the operations that are described below. Refer to the *Kernel Programming Guide* for information about how these commands are coded.

AQUEUE
enqueue an asynchronous read or write operation (called by **aread**(2) or **awrite**(2))

AQUEUE_INIT
prepare an asynchronous read or write operation for enqueuing (called by **arinit**(2) or **awinit**(2))

ACANCEL
cancel a pending asynchronous read or write operation (called by **acancel**(2), **exec**(2), and **exit**(2))

AQUEUE_TERM
free up resources that were used for a previous asynchronous read or write operation (called by **arwfree**(2), when the areq(D4X) structure is being reused for a new asynchronous I/O operation, when process exits, etc.)

areq pointer to the areq(D4X) structure for this operation

DESCRIPTION

The **aio** routine is used to initiate asynchronous read and write operations for character devices. Most control for an asynchronous I/O transfer comes from the user-level process; the driver's **aio** routine is coded to accept the information passed by the user-level program.

RETURN VALUES

The value returned from the **aio** routine varies with the value of the *cmd* argument:

AQUEUE_INIT	0	successful initialization
	EAGAIN	insufficient resources

	ENODEV	asynchronous I/O not supported for this particular device or transfer parameters
	ENXIO	illegal request
AQUEUE	0	successful queuing
	EAGAIN	insufficient resources
	ENODEV	asynchronous I/O not supported for this particular device or transfer parameters (will cause synchronous emulation if **fcntl**(2) set the F_SETAIOEMUL flag on the file descriptor)
	ENXIO	device error before transfer starts
	−1	the operation has been terminated by the driver with a call to **comp_aio**(D3X)
ACANCEL	ACANYES	request has been canceled
	ACANNOT	request is in progress; cannot be canceled
	ACANNIP	request has finished; cannot be canceled

The **aio** routine returns values that the generic asynchronous I/O code in the kernel uses to determine whether or not the I/O transfer was queued successfully. For the AQUEUE_INIT and ACANCEL commands, any error code is returned to the system call that initiated the I/O request (**arinit**(2), **awinit**(2), or **acancel**(2)).

For the AQUEUE command, the base-level routine has already committed to making an asynchronous return to the user. An error code from the driver is used by the base level of the driver to perform a **comp_aio**(D3X) to pass the error code back to the user by writing it to the **rt_errno** member of the `aiocb`(4) structure.

- If the driver returns a 0, it indicates that the driver has accepted the operation and will call **comp_aio** itself when the transfer is completed.

- When **aio** is called through the file system, the driver may have already called **comp_aio** before returning to the base level. In this case, the −1 return is used to notify the base level that the operation is no longer in progress.

- The −1 return is also used by the file system code if the offset is at end-of-file; in this case, **comp_aio** will have been called to indicate that there was no error and the byte count will have been set to zero.

DEPENDENCIES

Drivers using the **aio** routine must be configured as character special devices and identified as having an asynchronous I/O handler.

SEE ALSO

KPG, "Miscellaneous I/O Operations"
intr(D2X), **comp_aio**(D3X), **comp_cancel_aio**(D3X), areq(D4X)
aread(2), **awrite**(2), aiocb(4)

NAME

close, bclose, cclose – cease access to a device

SYNOPSIS

```
#include "sys/file.h"
#include "sys/open.h"

prefixclose(dev, flag, otyp)
dev_t dev;
int flag;
int otyp;
```

The synopses of **bclose** and **cclose** are the same as for **close**.

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 **open**(D2X) 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. For drivers installed with full semaphoring, the **close** routine is called in response to every **close** of the device; for drivers installed under one of the compatibility modes, the **close** routine is called only on the last **close** of the device, except when **close** is called with *otyp* set to OTYP_LYR. 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_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 both block and character devices.

OTYP_MNT close (unmount) a file system

OTYP_SWP close a swapping device

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. Every driver should have a **close** routine, although the routine may be empty. If the device was opened with a **bopen** or **copen** routine, then the corresponding **bclose** or **cclose** routine must be used to close the connection.

A device may be opened simultaneously by several processes and the **open** driver routine called for each **open**. In drivers installed under the compatibility modes, the kernel calls the driver **close** routine when the last process using the device issues a **close**(2) call or exits. In drivers installed as fully semaphored, the kernel calls **close**(D2X) for every **close**(2) system call.

The **close** routine may perform the following activities:

- ❑ deallocate buffers for private buffering scheme
- ❑ unlock an unsharable device (that was locked in the **open** routine)
- ❑ flush buffers
- ❑ notify device of the close
- ❑ issue **cintrelse**(D3X) to release connected interrupt structure

If an error occurs during **close**, **close** should test the **u.u_error** member of the user(D4X) structure to ensure that its value is zero (i.e., it does not already contain an error message); if it is empty, set it to indicate the error, but do not change the value if it already contains an error message. See the *open.h* file for more information.

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 deallocating resources and cleaning up data structures, as appropriate.

close in Fully–Semaphored Drivers

In drivers installed as fully semaphored, **close**(D2X) is called in response to every **close**(2) system call issued against the device, in order to avoid race conditions between **open** and **close** operations. If the driver needs to perform some tasks only on the last **close**, the driver should use a counter, as in the following example.

```
/* There is an iobuf structure for each device */
/* in this driver. Other drivers may use different */
/* data structures. */

extern struct iobuf xx_iobuftab[];
#define opncnt   io_s8

xx_init()
{
             ⋮
     initsema(xx_opn_sema, 1, 0);
     for (dp = xx_iobuftab;
          dp < &xx_iobuftab[xx_max_dev]; dp++) {
                dp->opncnt = 0;
     }
}
xx_open(dev, flag, otyp)
dev_t dev;
int   flag;
int   otyp;
{
             ⋮
     set up dp to point to the iobuf for this device
             ⋮
     psema(&xx_opn_sema, 0);
     dp->opncnt++;
     vsema(&xx_opn_sema, 0, 0);
}
xx_close(dev, flag, otyp)
dev_t dev;
int   flag;
int   otyp;
{
             ⋮
     code to be performed on every close
     set up dp to point to the iobuf for this device
             ⋮
     psema(&xx_opn_sema, 0);
     if (--dp->opncnt) != 0) {
          vsema(&xx_opn_sema, 0, 0);
          return;              /* not last close */
     }
             ⋮
     code to be performed only on last close
             ⋮
     vsema(&xx_opn_sema, 0, 0);
}
```

close in TTY Drivers

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

NAME

dump – save core image after a system panic

SYNOPSIS

prefix`dump( )`

DESCRIPTION

The **dump** routine is the driver interface for saving kernel memory images to supported block devices. **dump** is called by **unixcore**, which determines the dump device's major and minor numbers with **dumpinit**( ), then invokes the correct driver though the bdevsw(D4X) table with interrupts disabled (in other words, **dump** polls). The **dump** routine should start the dump at the kernel location labeled *firstmem*; dump the number of memory pages specified in kernel location *physmem*; and direct the dump to the device having the major and minor number specified at kernel location *dumpdev*.

The **dump** routine should include **cmn_err**(D3X) statements for error conditions that may arise, such as the inability to find the controller or device or too little space available on the dump device. The **dump** routine should also include the ability to reset and reinitialize the device and/or its associated controller following a double bus fault or any other condition that may leave the controller in a nonfunctional state.

DEPENDENCIES

Drivers supplying the **dump** routine must be configured as block special devices with a **dump** handler.

The device number for the dump special device file, */dev/dump*, must correctly specify the intended dump device specified by the system devices entry in **sysgen**(1M); this device is usually the system *swap* device. During system initialization, a script in */etc/rc2.d* copies the core image and associated bootable kernel image to the */usr/dumps* directory.

NAME

init – initialize a device

SYNOPSIS

prefix`init()`

DESCRIPTION

Every driver should have an **init**(D2X) routine, although some have nothing to initialize and others defer initialization to the **open**(D2X), **bopen**, **copen**, or **ioctl**(D2X) routine. In most cases, it does not matter if variables are zeroed in an **init** or an **open** routine. On the other hand, the system should be informed at the time of initialization if, for example, a disk drive is offline. Drivers that use kernel semaphores and spin locks should initialize them in an **init** routine so that the semaphores are associated with the appropriate data structures and initialized to the appropriate value when the system is booted.

Use **init** to execute functions when the computer is first brought up; use **open**, **bopen**, **copen**, or **ioctl** to execute functions after the operating system is started, file systems are mounted, and interrupts are enabled. The choice of routines to use for initialization should be made in consideration of the following:

- **init** cannot be used for any initialization that requires interrupts to be enabled because interrupts are disabled at the time the **init** routines execute.
- **init** must be used to initialize driver-specific kernel structures, in other words, structures other than the standard structures documented in Section 4.
- Driver initialization takes time. Often it is preferable to slow the system initialization time to avoid having the first user-level process that tries to access the device absorb the initialization overhead. If the driver uses the **init** routine or if a process called by */etc/inittab* calls the **ioctl** or **open** routine, all initialization will be done when the first application program attempts to access the device.
- Once memory is allocated for the driver, it is unavailable to other system processes, even if the driver is not using it. For infrequently used devices that do not require optimum performance, it may make sense to allocate kernel resources only when the device is actually being used. In this case, resources can be allocated in the **open**(D2X) routine and freed in the **close**(D2X) routine.
- Drivers for local bus boot devices must use the **init** routine.

In the following pseudocode for a software driver, the initialization processing required is minimal. Some memory must be allocated and initialized, and a warning must be issued if the allocation fails. The pseudocode example is listed in three sections, which are referenced by the section headers below to indicate the lines that are being explained.

```
(1) init(dev)
        if (memory can be allocated)
            allocate memory
            initialize memory

(2) initialize semaphores (initsema(D3X))
        semaphores for exclusive access of resources
        semaphores for sleep/wakeup functionality
    initialize spin locks (initlock(D3X))

(3) if initialization is successful
        print informational message
    else
        print warning message
```

Memory Allocation (1)

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

Semaphore Initialization (2)

The initialization routine for the driver must initialize all driver-specific kernel semaphores and spin locks:

- ❑ use **initlock**(D3X) to initialize a spin lock to 0 (unlocked)
- ❑ use **initsema**(D3X) to initialize a blocking semaphore to 0 (the first will decrement the value to −1 (blocked))
- ❑ use **initsema** to initialize an exclusionary semaphore to the number of resources available

Remember that all **psema**(D3X), **cpsema**(D3X), and some **vsema**(D3X) calls to a particular semaphore must use the same flags. So, if your driver must sometimes block in an interruptible state and sometimes in an uninterruptible state, you must initialize two blocking semaphores. Refer to the

Kernel Programming Guide for more discussion about using kernel semaphores and spin locks.

Messages (3)

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

```
cmn_err(CE_WARN,"prefix_init: sptalloc cannot allocate %d buffers", BUFS);
```

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

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

A working driver for a hardware device (for example, a disk drive) often requires a more complicated **init** routine than the one shown in the pseudocode above. The additional processing required may include some of the following:

- ❑ Confirm that the devices under the control of the driver are online.
- ❑ Check for the correct number of subdevices.
- ❑ Set each device's interrupt vector to correspond to the system's interrupt vector table.
- ❑ Set the virtual-to-physical address translation.
- ❑ Set device-specific parameters to default values. These parameters include values for the number of tracks, cylinders, and sectors.
- ❑ Download executable code to the controller. Controllers for many devices have their own processors and memory and are referred to as intelligent devices. The executable code downloaded to the controller is sometimes called pumpcode.

CAVEATS

init must never call kernel functions that issue the **sleep**(D3X), **psema**(D3X), or **vsema**(D3X) functions or functions that access the `user`(D4X) or `proc`(D4X) structure. Initialization activities that require access to these functions should be done in an **open**(D2X) or **ioctl**(D2X) routine.

NAME

intr – process a device interrupt

SYNOPSIS

```
void prefixintr(subvec)
int subvec;

int prefixintr(subvec)
int subvec;

void prefixintr()
int prefixintr()
```

ARGUMENTS

subvec indicates which controller associated with the driver generated the interrupt. This parameter can be omitted if only one device can generate the interrupt; refer to page 2–20.

DESCRIPTION

The **intr** routine is the standard interrupt-handling entry-point routine. It is used to handle interrupts that are generated by devices that have only one function and allow a unique vector to be assigned for the device. This assignment can be made through hardware (such as selecting a jumper) or through software (in which case, the REAL/IX Operating System handler sets up the hardware appropriately).

The **intr** routine is entered when a hardware interrupt is received from a driver-controlled device. It processes job completions, errors, changes in device status, and unexpected interrupts for both block and character drivers. The contents of the routine depend on the device it controls.

Devices with different interrupt capabilities and requirements can be implemented on the REAL/IX Operating System by implementing alien handlers and multiple handlers. For instructions about how to use these alternative interrupt-handling mechanisms, refer to the *Kernel Programming Guide* and the *Driver Development Guide*.

Devices that Generate One Interrupt

Simple interrupt-generating devices generate only one interrupt. The REAL/IX Operating System takes this style as its basic model of how devices work, but allows extensions to this model to allow for the many possible alternatives.

The **intr** routine is the normal interrupt routine for a driver. Because many similar devices, each of which generates just one interrupt, may be configured, a parameter is passed to the **intr** routine. This parameter allows the **intr** routine to determine the device that caused the interrupt. Refer to "The Interrupt Routine Argument" on page 2–20 for more information. Normally the **intr** routine is of type **void**, so there is no need to return a value to the interrupt envelope routine.

When an interrupt occurs, control is passed to an envelope routine that performs any necessary housekeeping (such as saving CPU registers or passing the appropriate parameter to the **intr** routine) and performs any actions required for the driver's synchronization method. Each synchronization method requires some different considerations in the interrupt routine; these are discussed later.

The system automatically generates the interrupt envelope for the device. When using alien handlers, you can write your own interrupt envelope; refer to the *Kernel Programming Guide* for more information.

The specific content of the **intr** routine is determined by the needs of the device, but it usually contains some combination of the following functionality:

- ❑ If an argument is supplied, interpret it to determine the source of the interrupt.
- ❑ Determine the cause of the interrupt.
- ❑ If appropriate, notify associated user-level processes of the condition signaled by the interrupt. Refer to page 2–20 for information about handling job completion interrupts, and page 2–22 for information about using connected interrupts to notify the user-level process.

  The **send_event**(D3X) kernel function can be used to post an event to the associated user-level process. It may also be appropriate to post a signal with **psignal**(D3X), **psignalcur**(D3X), **psignalval**(D3X), or **signal**(D3X).
- ❑ If the interrupt reflects a change in device status, record any necessary details.
- ❑ If the interrupt is due to some intermediate stage in a sequence, perform whatever action is required to continue. For example, certain I/O devices require that characters be sent to the device individually, in which case an interrupt may request the next character from an output buffer.
- ❑ If the condition signaled by the interrupt allows another operation to start, search the driver queues for a queued operation and start it.

❑ If the interrupt indicates a device error, process it appropriately.

❑ Handle stray or spurious interrupts gracefully. Diagnostics may be kept, but the system should not be halted for stray interrupts except during debugging.

❑ If necessary, update statistics as required by the driver.

Devices that Generate More Than One Interrupt

The basic interrupt-handling model of the REAL/IX Operating System must be extended when a device generates more than one interrupt. The usual method is to use the **intr** routine to handle whichever interrupt is most likely to report I/O completions and to use alien handler routines to deal with the remaining interrupts. Refer to the *Kernel Programming Guide* for details.

If a device generates several different interrupts that form a contiguous range, it is possible to route all of these interrupts to the **intr** routine. The interrupt vectors size field in the driver screen must be set to the number of contiguous vectors multiplied by 4. Refer to the *Driver Development Guide* for details.

The following guidelines can help you decide whether to route all of a range of contiguous interrupts to a single **intr** routine:

❑ The **intr** routine cannot readily distinguish the source of the interrupt, because the *subvec* parameter will be identical for all interrupts within the range. Consequently, additional processing must be done at the interrupt level, thus degrading the system's interrupt latency.

❑ Drivers installed under the compatibility modes cannot support alien handlers, although they can support an **intr** routine that handles a range of contiguous interrupts.

Interrupt Routine Restrictions

Keep the following restrictions in mind when developing an interrupt routine:

❑ Interrupt routines must not set any fields in the `user`(D4X) structure, because the process running when the interrupt occurs may not be the process that initiated the I/O operation.

- ❑ For the same reason, interrupt routines must not call any functions that block (such as **sleep**(D3X) or **psema**(D3X) or functions that call **sleep** or **psema**). The D3X manual pages identify the functions that can be called from the interrupt level.

- ❑ For drivers installed under one of the compatibility modes, **spl***(D3X) functions must not drop the processor execution level below the level set for the interrupt routine. Doing so can corrupt the stack.

- ❑ There may be cache coherency considerations. Refer to the *Kernel Programming Guide* for information about memory management.

The Interrupt Routine Argument

The **intr** routine takes one (optional) argument, which indicates the controller that generated the interrupt. By passing an argument, one interrupt routine can handle the many different interrupt vectors associated with the many devices that may be controlled by the one driver. The argument, *subvec*, is the result of the controller number multiplied by the number of devices per controller. It usually indicates the minor number of the first subdevice on the controller. For instance, if a subdevice on controller 0 issues an interrupt, and the controller supports two subdevices, *subvec* would be 0 (controller 0 times 2 subdevices equals 0). If controller 1 (with the same configuration) issues an interrupt, *subvec* would be 2.[1]

Note that not all interrupt handlers receive or need parameters. If it is certain that a driver will never support more than one device, the *subvec* argument is redundant (it will always have the value 0). In this case, the driver can be **sysgen**ed so that no argument is passed, which saves a couple of machine instructions per interrupt.

Handling Job Completion Interrupts

For job completion interrupts, service depends on the requirements of the application:

- ❑ For I/O operations initiated by the **read**(D2X), **write**(D2X), **strategy**(D2X), or **mbstrategy**(D2X) entry-point routines, the interrupt handler routine unblocks any base-level process waiting on the interrupt completion. For example, when a disk drive has transferred information to the host to satisfy a read request, the disk drive

[1] The order in which the *subvec* number is assigned is determined by the alphabetical order in which the devices are listed on the **sysgen**(1M) item screens. This is determined by the contents of the left column on that screen (i.e., the board description).

generates an interrupt. The CPU acknowledges the interrupt and calls the disk driver's interrupt routine. The driver interrupt routine then unblocks the process waiting for data, which conveys the data to the user.

The function issued to unblock is determined by the function used to block:

If the driver blocked with:	**intr** *unblocks with:*
psema(D3X)	**vsema**(D3X)
iowait(D3X)	**iodone**(D3X)
preiowait(D3X)	**iodone**(D3X)
sleep(D3X)	**wakeup**(D3X)

- For I/O operations initiated by the **aio**(D2X) entry-point routine, the base level of the driver is not blocked awaiting completion of the I/O operation. Rather than unblock a process, the interrupt routine issues a function that updates the `areq`(D4X) structure:

    - **comp_aio**(D3X) is used if the I/O operation completed.

    - **comp_cancel_aio**(D3X) is used if the I/O operation was canceled with an **acancel**(2) issued by the user-level process.

  Refer to the *Kernel Programming Guide* for more detailed information about coding the driver to use asynchronous I/O.

The following pseudocode illustrates how the interrupt routine is coded to handle job completion interrupts for a block device:

```
drivintr(dev)
{
        identify the subdevice that interrupted
        find the buffer associated with that device and remove it from queue

        if (some_error_condition) {
                set error indicators in the buffer header
        }
        iodone(bp);
        if (entries_remain_on_device_queue) {
                start up next request on queue
        }
}
```

Servicing Interrupts with Connected Interrupts

A number of devices used for such realtime applications as process monitoring and control receive interrupts intended to notify the appropriate user-level process of an external event rather than to signal the completion of an operation requested by the base level of the driver. Rather than notifying the base level of the driver of the interrupt, the interrupt-handling routines of such drivers use the connected interrupt mechanism to notify the user-level process of the interrupt.

The following gives an overview of the coding required to use the connected interrupt mechanism.

1. The user-level process populates a `cintrio`(4) structure that determines how connected interrupts will be handled, then uses the CI_CONNECT command to **ioctl**(2) to connect the driver.

2. The driver executes the **cintrget**(D3X) function to establish a *cid* (connected interrupt ID) that is used to identify this connected interrupt in all subsequent connected interrupt kernel functions. **cintrget** also populates the `cintr`(D4X) structure with information passed through the driver from the `cintrio` structure.

3. If the interrupt is the type to be handled with a connected interrupt, the driver's **intr** routine calls the **cintrnotify**(D3X) function or the **CINTRNOTIFY**( ) macro. If desired, **cintrnotify** can also pass a 32-bit data item, which will be posted to the user-level process with the event. The operating system checks the appropriate `cintr` structure for the notification method:

   - If the method is CINTR_EVENTS, the system posts an event to the user process.

   - If the method is CINTR_POLL, the interrupt handler increments the location pointed to by ***ci_polloc**; the user-level program will poll that location and learn of the interrupt. Note that the ***ci_polloc** pointer can also be used to return a 32-bit data item to the user.

The connected interrupt mechanism also includes facilities to allow the user-level process to change the notification method as well as to determine whether more than one connected interrupt for the structure/process can be processed at a time. Also, the `cintrio` structure includes one member that can be customized for the needs of the driver.

Refer to the *Kernel Programming Guide* for a code example of a driver that uses connected interrupts and the associated user-level process that accesses it. Additional examples are in the */usr/examples/pio* directory.

Writing Interrupt Routines for Intelligent Boards

Intelligent boards provide the facility to share a queue with the interrupt handling routine and can take on some responsibility for moving data to and from the device. By using queues in memory, the number of interrupts that need to be requested by the device can be reduced. In contrast, devices controlled by unintelligent boards, frequently TTY devices, must interrupt the CPU each time a character is sent or received. The exact method whereby the host talks to an intelligent board will be determined by the board itself, but the following steps are typical:

1. The driver's **init** routine formats an area of memory as a queue with pointers to the beginning and end of the queue. The type of queue is defined by the controller.

2. When this queue is set up, **init** notifies the board by writing a startup message directly into the hardware. Typically, until this is done, the board waits for "standalone" commands sent by the driver that poll an area on its internal memory.

3. The driver first formats a command buffer, then writes one word into the board memory to indicate that a command has been issued. That command contains pointers to the places in memory where the board should look for jobs that are associated with this device, such as the job request queue and the job completion queue.

4. The driver writes a job in this buffer, updates the load pointer to indicate that there is a job waiting, and signals the hardware by either a control status request (CSR) bit or through some mechanism on the board that causes it to look at the job queue.

5. The interrupt handler must also update the status information, set/clear flags, set/clear error indicators, and so forth to complete the handling of a job.

6. When the routine finishes, it should advance the unload pointer to the next entry in the completion queue.

The advantage of this protocol is that it avoids memory contention between the hardware and the software because the driver updates the load pointer and the hardware updates the unload pointer when it gets the job. When the job is completed, the hardware puts a job in the queue (assuming there is room), updates the load pointer, and sends an interrupt to indicate that the job is completed. The driver's **intr** routine checks the data structures to determine which of the devices interrupted and how many jobs are in the queue.

Shared Driver/Device Structures

Structures shared between a driver and a device present some specific difficulties that must be addressed by the interrupt routine:

- □ Information in the shared structure may be updated at any time by the device. The structure must be monitored by the interrupt routine. **spl***(D3X) functions cannot be used to prevent the device from changing a structure shared between a driver and hardware, even if the driver is installed under CPU affinity. The type of protection depends on the controller firmware, but is usually accomplished in one of three ways:

  - ▪ Define a scheme so the driver and controller access different portions of the structure.

  - ▪ Use an interrupt to "lock out" the controller until the driver indicates that it is done.

  - ▪ If the hardware is smart enough to examine a flag in the control register or memory location to determine if it is safe to update the structure, set up a protocol on which the driver and hardware agree. (The protocol is usually defined by the hardware.)

- □ Additional interrupts may occur, signaling the completion of jobs previously passed to the hardware while the interrupt routine is processing a previous interrupt. The most efficient way of handling this is to have a loop that compares the load and unload pointers on the completion queue.

  A job placed on the queue cannot be seen or acknowledged by the driver code when the driver is in the interrupt routine. What the driver can see is that the load pointer has moved. Using this indicator, the driver can handle the new job. This presents an additional problem: the driver interrupt routine must be prepared to unload more than one job from the queue.

- ❑ An interrupt is normally requested after the last request is processed. Because this interrupt is issued by the last request, the last job may have already been unloaded. This interrupt has no job associated with it, and the interrupt routine must recognized that this interrupt is not an error condition.

  One way to ensure that the last interrupt is a holdover with no work attached to it is to keep a count of the number of jobs outstanding. The counter is incremented when the job is put on the request queue and decremented in the interrupt routine when the job is removed from the queue. Generally, this information may be kept in a separate data structure used for job status for each device or controller.

Interrupt Handlers for Major Device Semaphoring

Interrupt routines for drivers that are semaphored on the major device number usually do not need to be rewritten to run on the REAL/IX Operating System, although the interrupts are handled a bit differently than for fully semaphored drivers. When a driver is installed with major device semaphoring, a semaphore is assigned to the driver code itself. When a device interrupt is received, the interrupt entry code issues a **cpsema**(D3X) function call to see if it can lock the semaphore.

- ❑ If **cpsema** finds the semaphore locked, a flag bit[1] is set to defer the interrupt. This flag is checked when the semaphore is unlocked to determine if the interrupt routine needs to be called.

- ❑ If **cpsema** is successful, the flag in the switch table for the subdevice is cleared and the **intr** routine is called to service the interrupt. On return from the driver, the interrupt envelope code releases the semaphore.

Major-device semaphoring prevents a base-level routine from being preempted by another instance of itself executing on a different processor and ensures that an interrupt-handling routine will not occur during execution of the base-level routine. Depending on the number of subdevices serviced by the driver, it may be possible to improve driver performance by using minor device semaphoring or rewriting the driver to use the kernel semaphoring functions, both of which reduce contention for the device semaphore.

[1] A bit in the **d_unit** field of the `semdrivs`(D4X) structure pointed to by the **d_sems** member of the switch table.

Note that interrupts are delayed by setting a single flag. If multiple interrupts happen asynchronously, they may result in a single call to the interrupt-handling routine. The flag bit that is set is determined by the parameter that is to be passed to the **intr** routine. There are 32 flags, numbered from 0. Therefore, an interrupt handler using major device semaphoring is limited to configurations that do not require parameter settings of 32 or greater.

Major-device semaphoring assumes that an interrupt can be "ignored" until the base-level routine exits. Drivers for devices that continue to assert the interrupt even after the hardware interrupt acknowledge cycle may not be able to defer the interrupt. The easiest way to determine whether this option can be used is to install the driver on an otherwise quiet system and try it. If the system does not hang, the device supports the functionality required to use major device semaphoring; if the system hangs, the driver must be rewritten to use the kernel semaphore functions or to be hard-assigned to one CPU.[1]

Interrupt Handlers for Minor Device Semaphoring

The interrupt portion of the driver for devices semaphored on the minor number must be written differently than interrupt routines for drivers installed under any other kind of semaphoring. The interrupt handling functionality is put into the **serv**(D2X) routine, and the **intr** routine is written to determine the subdevice that caused the interrupt, as in the following example.

[1] This is the CPU affinity compatibility mode, which is not supported on all machines. Refer to the Release Notes shipped with your system.

```
    ⋮
extern struct semdrivs xxsems[];
xxxxintr(minor_dev)
int minor_dev;
{
    struct semdrivs *sp;
    dev = some_function_of(minor_dev, device status ...);
    sp = &xxsems[dev];
    spsema(&sp->d_lock);
    if (rcpsema(&sp->d_sema, SEMRTBOOST)) {
         sp->d_unit = 0;
         svsema(&sp->d_lock);
         *++c.c_istk_ndx = &sp->d_sema;
         xxserv(dev);
         --c.c_istk_ndx;
         vsema(&sp->d_sema, 0, SEMRTBOOST);
    } else {
         sp->d_unit = 1;
         svsema(&sp->d_lock);
    }
}
    ⋮
```

The **intr** routine does a **cpsema**(D3X) to try to lock the subdevice. If **cpsema** is successful, it calls the **serv**(D3X) routine to service the interrupt; otherwise, it sets the **d_units** bit in the semdrivs structure to mark that an interrupt is deferred and waits to service the interrupt until the base level of the driver exits (with, for instance, **sleep** or **delay**), at which point the **intr** routine calls **serv** to handle the interrupt. Interrupts are handled similarly for minor-device semaphoring and major-device semaphoring; the recoding of the interrupt handler for minor-device semaphoring is necessary to determine the subdevice that caused the interrupt so the system knows which semaphore to lock.

In addition, you must add spin locks (with **spsema**(D3X) and **svsema**(D3X)) in the interrupt-level routines to protect any data structures or device registers that are shared by two or more subdevices.

SEE ALSO

KPG, "Interrupts"
DDG, "Porting Drivers"
serv(D2X), `semdrivs`(D4X)

NAME

ioctl – control a character device

SYNOPSIS

```
prefixioctl(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. The kernel does not interpret the command type, so a driver is free to define its own commands (within the limitations defined in "REAL/IX I/O Control Commands" on page 2–33).

termio(7) specifies the command types that must work for AT&T terminal drivers.

cintrio(7) specifies the command types used with the connected interrupt mechanism.

Create a unique identifying command so your driver can ascertain that a correct command has been received. This should be done to guard against misuse by users. Be sure to comment the commands you create.

arg passes parameters between a user-level 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*).

Refer to the *flag* argument description of the **open**(D2X) routine for other values for the **ioctl** routine's *mode* argument.

DESCRIPTION

The **ioctl** routine provides character-access drivers with an alternative entry point that can be used for almost any operation other than a simple transfer of characters in and out of buffers. Most often, an I/O control command is used to control device hardware parameters and to establish the protocol used by the driver for processing data.

After the user-level program opens a special device file, it can pass I/O control command arguments. The kernel looks up the device's file table entry, determines that this is a character device, and looks up the entry-point routines in cdevsw(D4X). 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 an I/O control command, so it is up to the user program and the driver to agree on what the arguments mean.

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

- □ implement terminal settings passed from **getty**(1M) and **stty**(1)
- □ format disk drivers
- □ implement a trace driver for debugging network drivers
- □ clean up character queues
- □ recalibrate a robotic device
- □ control process I/O equipment (analog-to-digital, digital-to-analog, digital I/O)

Because the kernel does not interpret a command that defines an operation, a driver is free to define its own commands. Note that both connected interrupts and asynchronous I/O use I/O control commands; applications using either of these mechanisms must use different I/O control commands for application-specific purposes.

Drivers that use an **ioctl** routine typically have a command to read the current I/O control command settings and at least one other command that sets new settings. If necessary, you can use the mode argument to determine if the device unit was opened for reading or writing 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 download data into the driver itself (not through the driver to the hardware). In this case, the operation argument is a pointer to a buffer of an appropriate size that contains the data. The buffer itself should be set up by a user-level process or daemon.

Two steps are required to implement I/O control commands for a driver:

1. Define the I/O control commands and the associated values in the driver's header file.

2. Code the driver **ioctl** routine to define the functionality for each I/O control command in the header file.

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

Defining I/O Control Command Names and Values

The I/O control command name is passed as the second argument (*cmd*) to the driver **ioctl** routine. It should be defined, along with an integer value that is actually passed, in the driver's header file.

The I/O control command name and value can be defined in the driver code itself, but this is not recommended. If I/O control commands are defined in a header file, the user program and the driver can both access the same definitions to ensure that they agree about what each I/O control command value represents.

The I/O control command name is traditionally an uppercase alphabetic string. This alphabetic name can be a mnemonic. You should try to keep the values for your I/O control commands distinct from other I/O control command values on the system. Each driver's I/O control commands are discrete, but it is possible for user-level code to access a driver with an I/O control command that is intended for another driver, which can lead to serious consequences, such as if it meant to pass "drop carrier on a communication line," but instead sends the argument to a disk where it is interpreted as "reformat driver." Permissions can be set to prevent most such events, but the more unique your I/O control command values are, the safer you are. Each driver has up to 2^{32} values that can be passed as an integer, so it is quite possible to avoid using numbers that are already in use.

Various schemes are legal for assigning values to I/O control command names. The most straightforward is to use decimal values. For example:

```
#define COMMAND1   01
#define COMMAND2   02
```

Similarly, you can assign hexadecimal numbers as values:

```
#define COMMANDA   0x0a
#define COMMANDFF  0xff
```

The drawback to these methods is that one quickly gets an operating system that contains several instances of each I/O control command value, with the inherent risks discussed above.

A common method for assigning I/O control command values that are less apt to be duplicated is to use a shift-left-8 scheme. For instance:

```
#define COMMAND10    ('Q'<<8|10)
#define COMMAND11    ('Q'<<8|11)
#define COMMAND12    ('Q'<<8|12)
```

Alternatively, the shift-left-8 scheme can be defined as a constant, which is then used for the I/O control command definitions. For example:

```
#define ROTA         ('q'<<8)
#define COMMAND23    (ROTA|234)
#define COMMAND25    (ROTA|254)
```

An alternative coding style is to use enumerations for the command argument, which allows the compiler to do additional type checking, as in the following:

```
typedef enum {
      XX_COMMAND10 = 'Q'<<8 | 10,
      XX_COMMAND11 = 'Q'<<8 | 11,
      XX_COMMAND12 = 'Q'<<8 | 12,
} xx_cmds_t;
```

REAL/IX I/O Control Commands

Before defining I/O control commands, check any system header files you `#include` to ensure that the I/O control command values you are defining are not already used. In particular, the connected interrupt and asynchronous I/O mechanisms use the I/O control commands listed Table 2–3.

Table 2–3. System–Defined I/O Control Commands

Command	Value	Header File	Description
AIOGETREQ	'A'<<8|0x00	aio.h	get information
CI_CONNECT	'I'<<8|1	cintrio.h	connect to device interrupt
CI_UCONNECT	'I'<<8|2	cintrio.h	disconnect from device interrupt
CI_SETMODE	'I'<<8|3	cintrio.h	set modes of device interrupt
CI_STAT	'I'<<8|4	cintrio.h	get status of device interrupt
CI_ACK	'I'<<8|5	cintrio.h	acknowledge device interrupt

For an example of how an **ioctl** routine is coded to support connected interrupts, see the **avme9510** or **pccclk2** driver supplied under the */usr/examples/pio* directory or the */usr/examples/pcc* directory, respectively. This same routine illustrates how to implement "peek and poke" functionality using an **ioctl** routine.

Coding the ioctl Routine

The header file for the driver should define all I/O control commands and structures used. While this information can be included in the driver itself, this is not recommended. The general shape of the header file that defines the I/O control commands and an **ioctl** routine is illustrated below.

```
#define EXAM     ('E'<<8)
#define COMMAND1 (EXAM|01)
#define COMMAND2 (EXAM|02)
#define COMMAND3 (EXAM|04)

struct send_to_device
   {
   int flags;
   char setup[64];
   };

struct receive_from_device
   {
   int flags;
   char current_status[64];
   };
```

Sample I/O Control Command Header File

The **ioctl** routine is coded with instructions on the proper action to take for each I/O control command. Generally, a driver's **ioctl** routine consists of a **case** statement for each I/O control command that identifies the required action. The command passed to a driver by a user process is an integer value that is associated with an I/O control command name in the header file.

The **case** statement should have a default case to return an error value if the driver is called with an unknown I/O control command.

The **ioctl** routine that is associated with the header file in the previous example looks like the following:

```
#include example.h

xxioctl(dev, cmd, val, flag)
int dev;
int cmd;
caddr_t val;
int flag;
{

   switch(cmd)
   {
   case COMMAND1;
   /* send new status setup to device */
      senddev((struct send_to_device *) val);
      return;

   case COMMAND2:
   /* get current status from device */
      recdev((struct receive_from_device *) val);
      return;

   case COMMAND3:
   /* return number of devices */
      *val = SUBDEVICES;
      return;

   default:
      u.u_error = EINVAL;
      break;
   }
}
```

Sample I/O Control Command Routine

DEPENDENCIES

Drivers using the **ioctl** routine must be configured as character special devices with an ioctl handler.

Drivers that support asynchronous I/O must supply an interface to the system-defined AIOGETREQ I/O control command (refer to Table 2-3). The **ioctl** routine associated with such a driver should include a **case** statement for AIOGETREQ similar to the case statements shown in the example above.

NAME

mbstrategy – handle multiple block device input and output

SYNOPSIS

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

ARGUMENTS

bp pointer to the address of an instance of the buffer header data structure defined in the system header file *buf.h* (refer to buf(D4X))

DESCRIPTION

mbstrategy(D2X) is very similar to the **strategy**(D2X) routine. The major difference between them is that **mbstrategy** uses a chain of buffer headers to take advantage of any contiguity of disk blocks, using one operation to accomplish a data transfer instead of multiple calls to the **strategy** routine. The code controlling the buffer cache looks to see if the driver for a particular device supports multiple block I/O. If so, it combines what would have been several calls to the normal **strategy** routine into a single call to the **mbstrategy** routine.

The **mbstrategy**(D2X) entry-point routine is unique to the REAL/IX Operating System. Block drivers for disk devices or other devices that can be mounted as block special devices may optionally provide an **mbstrategy** routine to support multiple block I/O transfers.[1] This can, in many cases, improve overall system throughput.

mbstrategy routines must either perform the entire transfer specified or report an error. Error recovery is performed at a higher level in the kernel, where the failed **mbstrategy** call is transformed into a number of calls to the traditional **strategy** routine. The philosophy is to simplify the error-handling requirements on calls to **mbstrategy** on the assumption that they are infrequent and can be passed on to existing error-handling code.

As a result, there is no use of the residual byte count field to report partial transfers. If, for example, an **mbstrategy** routine is called with a buffer indicating a read at end of medium, the entire transfer is returned with B_ERROR set in the **b_flags** field. Contrast this with **strategy**(D2X) where the residual byte count is set to the initial transfer request but no error is reported.

The buffer header *bp* is the first in a singly linked list of buffer headers. The **b_chnnxt** field is the pointer to the next buffer in sequence or is null when

[1] At present, all SCSI devices are configured by default to support multiple block I/O. This feature can be enabled and disabled through **sysgen**(1M) on a system-wide basis or for individual devices. Tunable parameters are used to adjust the performance of the multi-block transfers. Refer to the *Kernel Programming Guide* for more information.

the last buffer header in the list is encountered. All information about the data transfer is contained in the fields of the buffer headers. Note that the data transfer specified by the buffer headers in the list will be for sequential blocks.

mbstrategy uses the following fields in the buf(D4X) structure of the first buffer header:

b_dev contains the major and minor number of the device where I/O is to occur.

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

b_bcount the number of bytes in the first data buffer.

b_un.b_addr a pointer to the first data buffer.

b_flags

- B_READ if set, this is an input operation. If not set, this is an output operation.
- B_CHAINED should always be set, marking this buf structure as an element in a list.
- B_CHNHEAD should always be set, marking this buf structure as the head of the list.
- B_ASYNC if set, indicates that the transfer is taking place asynchronously. There is no process that will be waiting specifically for the transfer to complete. This information is typically of no interest to the **mbstrategy** routine, only to the **iodone**(D3X) routine called when the operation is completed.
- B_PHYS should always be reset for this version of the REAL/IX Operating System.

b_error used to report any errors.

b_chnnxt a pointer to the next buffer header in a singly linked list.

b_start may be used to time I/O operations.

b_drivwksp a pointer to a workspace area that the driver may use. The workspace can be used to construct an array of `djntio`(D4X) structures to control the data transfer. The workspace size is given by the external variable *mbdjnt_size*. This is the number of `djntio` structures that can be contained in the workspace, minus one. Thus, the count is the number of useful structures that can be fitted in the area, assuming that an additional null entry is required to terminate the list. Include the file *sys/disjointio.h* for appropriate declarations.

mbstrategy will typically use the following fields in the `buf`(D4X) structure of buffer headers that are in the linked list:

b_bcount the number of bytes in the data buffer. Note that all buffers in the list will have the same size.

b_un.b_addr a pointer to the data buffer.

b_chnnxt a pointer to the next buffer header in the singly linked list. A null pointer implies that this buffer header is the last in the list.

In addition to those listed above, additional fields in the linked list of buffer headers will be set. Note that the system guarantees the settings of these fields; they do not need to be checked on a routine basis. If there is any consistency checking in the **mbstrategy** routine, any detected error will indicate a serious system fault that justifies the use of a system panic. The additional fields are:

b_chnhead points to the first buffer in the list.

b_flags

B_READ should be consistent with that of the equivalent flag in the first buffer.

B_CHAINED should always be set.

B_CHNHEAD should always be reset.

b_dev identical to **b_dev** in the initial buffer header.

b_blkno the **b_blkno** fields should always be sequentially ascending. Note that **b_blkno** is given in terms of physical block number, not logical block number. The physical and logical block numbers are related in a manner that depends on the block size. Each block is assigned a logical block number. The physical block number is equal to the logical block number multiplied by the block size and divided by 512.

All buffer headers in the list except for the first one will have the **b_chnhead** field set up to point to the first buffer header.

mbstrategy routines should not access the `user`(D4X) data structure because the process on whose behalf the transfer is to take place may not be the currently active process.

SEMAPHORE RAMIFICATIONS

Drivers providing an **mbstrategy** routine must be fully semaphored.

DEPENDENCIES

Drivers providing an **mbstrategy** routine must be configured as having both block and character special devices and identified in **sysgen**(1M) as having a multi-block strategy handler.

SEE ALSO

KPG, "Synchronized I/O Operations"
strategy(D2X), **intr**(D2X), `buf`(D4X), `djntio`(D4X)

NAME

open, bopen, copen – start access to a device

SYNOPSIS

```
#include "sys/file.h"
#include "sys/open.h"

prefixopen(dev, flag, otyp)
dev_t dev;
int flag, otyp;
```

The synopses of **bopen** and **copen** are the same as for **open**.

ARGUMENTS

dev device number (the unit number of the physical device being opened).

flag information passed from the user program; **open**(2) or **creat**(2) system call 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 file pointer to end of 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 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 write 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 both block and character devices.

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) manual page. 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, no other processes are using or sleeping on the device, then lock the device. An unsharable device is one that should be opened by one process at a time.

The kernel calls the driver **open** routine 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 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).

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 through **sysgen**(1M).

Additional **open** routine operation is dependent upon the device being opened. For example, the **open** routine for a removable media disk driver 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 wait on data terminal ready (DTR).

open is an entry-point routine for both block and character access. If you need separate functionality for block opens and character opens, use the **bopen** and **copen** entry points instead.

NAME

print – display a message on the system console during a block I/O operation

SYNOPSIS

```
prefixprint(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. The **print** routine permits the driver to expand device-dependent information (such as the device number) into meaningful error messages.

The **print** routine is used only for the block I/O transfers done by the **strategy** routine. In other cases, use the **cmn_err**(D3X) function to send messages to the console.

DEPENDENCIES

A driver using the **print** routine must be configured as a block device.

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.

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.

T_DISCONNECT send a command to the terminal controller to request that it disconnect a terminal device (tell it to drop the 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. For intelligent terminals that use the tty structure, the driver **proc** routine is called to update the device to the new parameters. 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 an XON 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. Refer to **termio**(7) for more 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 suspend output to the terminal because an XOFF character has been received. The driver **proc** routine should set the TTSTOP bit in **t_state** in the `tty` structure, and flush any input queues maintained by the driver.

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

```
&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 is not true in **t_lflag**), and then calls the device driver **proc** routine with the command flag T_SWTCH.

T_TIME notify the driver that delay timing for a break, carriage return, and so on, has completed.

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

T_WFLUSH clear the transmit buffer and output queue(s) of characters, and performs an implicit XON (T_RESUME).

proc(D2X)

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.

Note that **spl6**(D3X) is set when these flags are set.

DEPENDENCIES

This routine is used only by character drivers written in the TTY subsystem, which must be installed under one of the compatibility modes (CPU affinity, major-device semaphoring, or minor-device semaphoring).[1]

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
`tty`(D4X)

[1] Not all compatibility modes are supported on all machines. Refer to the Release Notes shipped with your system.

NAME

read – read data synchronously from a character-access device

SYNOPSIS

prefix`read(dev)`
`dev_t dev;`

ARGUMENTS

dev	device number

The following members of the `user`(D4X) structure are implicit arguments to the **read** routine:

u.u_base	address of the buffer in user virtual memory where the **read** data is to be found
u.u_count	byte count for the data transfer
u.u_ap	points to the original parameters of the **read**(2) system call
u.u_segflg	set to 0
u.u_fmode	copy of the **f_flag** member of the `file` structure (defined in *file.h*). The flag propagates the modes set in the **open**(2) request.
u.u_offset	current offset in the file

DESCRIPTION

When **read**(2) is executed, the driver initiates and supervises the data transfer from the device to the user data area. **read** is accessed through the character device switch table, `cdevsw`(D4X).

The **read** routine typically does the following:

- ❑ Validate the device number; if invalid, set **u.u_error** to ENODEV
- ❑ Initiate the data transfer:
  - For TTY drivers, use the **ttread**(D3X) function to do the transfer using the `tty`(D4X) structure to get a `cblock`(D4X) for buffering the transfer and update the `user`(D4X) structure. This is generally used for low-speed character devices.
  - For raw I/O on a block device, use the **physck**(D3X) and **physio**(D3X) functions to initiate the transfer. **physio** handles memory page locking to ensure that the pages impacted by the I/O are not swapped out and does the unbuffered I/O while maintaining the buffer header as the interface structure.

- For other character drivers, use the **copyin**(D3X) function to move the data from the user area to the kernel buffer area and from the kernel buffer area to the device. This transfer is done by pointing to the **u.u_base**,**, u.u_count**, and **u.u_segflg** members of the user(D4X) structure. If not using one of the system-supplied buffering schemes, the driver must set up its own buffering scheme; this is generally used with high-speed character devices such as network interface boards.

- □ Block on a semaphore with **psema**(D3X) to suspend execution until the I/O operation is complete. If the driver is installed under CPU affinity, major-device semaphoring, or minor-device semaphoring, you block with **sleep**(D3X).

- □ After the **intr**(D2X) routine unblocks the semaphore with a **vsema** (or **wakeup** if the driver blocked with **sleep**) signaling that the I/O operation is complete, the **read** routine must initiate a transfer of data from the kernel buffer area to user address space.

Return Values

On return from the driver, the following members of the user(D4X) structure are used to generate the return values for the **read**(2) system call:

u.u_error set if an error occurred during the I/O operation.

u.u_count set to the residual byte count (in other words, the amount, if any, of the requested transfer that could not be transferred). Set to 0 if all data was transferred.

In addition, the byte count parameter supplied by the user (pointed to, along with other parameters, by the **u.u_ap** member) may have been changed. The **read**(2) system call calculates the number of bytes transferred as the difference between the byte count parameter and the residual byte count in **u.u_count**. If, for example, the read is going to a block device and would extend beyond the limits of the device, the driver may scale down the request before passing it to a **strategy**(D2X) routine. There is no residual byte count from the scaled down request, but the transfer count returned from the system call has to reflect the reduced transfer size. This can be achieved by setting the byte count parameter to the lower value.

Read Routines that use physio(D3X)

Most devices that use block access also support raw or character I/O. Character I/O for a block device is also referred to as physical I/O because data bypasses the system buffer cache and is transferred directly from the device to in-core user memory space. The advantage of physical I/O is that data can be transferred more quickly and in larger quantities than with the system buffer cache, and kernel overhead is reduced by eliminating buffer handling. However, because physical I/O actually locks down portions of user memory and prevents it form being paged, overall system performance may be degraded. For this reason, physical I/O is used primarily for administrative and realtime functions where the speed of the specific operation is more important than overall system performance.[1]

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

If the data transfer is less than one page, **physio** can do the transfer directly between user address space and the device, avoiding the intermediary transfer into the kernel. Because I/O operations to devices must be made from physically contiguous pages (which are not guaranteed in user address space), for larger transfers, the driver must first call **dma_breakup**(D3X) to allocate a free buffer header from a pool of physical I/O buffer headers set by the tunable parameters NPBUF. These buffer headers are defined by the buf structure, but do not point to a specific address in the system buffer cache. Instead, the data pointer is assigned the location in user memory where the data transfer should come from or go to. This location is determined from the **u.u_base** member of the user structure. The **strategy** routine then uses this buffer header to control the I/O operations.

The following is typical job sequence for a physical I/O **read** operation. A **write** operation is similar, except that the **b_flags** member of the buf structure is set to B_WRITE instead of B_READ. The code that follows is an example **read** routine for a disk driver using physical I/O. The line numbers included in the following job sequence refer to the sample **read** routine.

1. The user program issues a **read**(2) system call to the kernel of the form "read 10,240 bytes from *character-special-file* to *virtual-address-*

[1] For example, when backing up a file system, completing the backup quickly is usually of greater concern than maintaining optimal system performance during the time allotted for backup operations.

N". The virtual address is a portion of user memory used to store user process data.

2. The kernel **read** routine started by the **read**(2) system call accesses the cdevsw(D4X) table to call the driver's **read**(D2X) routine.

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

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

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

6. The **physio** function then calls the **strategy** routine and issues a **psema**(D3X) to block[1] until the I/O operation is completed.

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

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

9. When the transfer is complete, the controller interrupts and the driver's **intr**(D2X) routine is entered. The **intr** routine uses the **iodone**(D3X) function to unblock the process that called the **physio** routine.[2] The **physio** function then updates information about the user(D4X) data structure, releases the buffer header, and eventually returns to the driver's **read** routine, which in turn returns to the kernel's **read** routine.

[1] **psema** is used only in drivers that are fully semaphored. Drivers installed under CPU affinity, major-device semaphoring, or minor-device semaphoring go to sleep (using the **sleep**(D3X) or **iowait**(D3X) function) on the address of the buffer header. Note that CPU affinity is not supported on all machines; refer to the Release Notes shipped with your system.

[2] **iodone** is used for all block drivers, whether or not they are fully semaphored. The function issues either a **vsema**(D3X) or a **wakeup**(D3X) function call as appropriate.

The following code example illustrates a **read** routine from a sample disk driver:

```
dskread(dev)
register dev_t dev;
{
     register unit                         /* disk controller ID */
     register unsigned char drv;           /* disk drive ID */
     register struct dskc *dskcp;          /* disk controller pointer */
     register struct dskpart *partpt;      /* pointer to partition info */
     register unsigned char part;          /* drive partition */

     unit = minor(dev);
     dskcp = &dsk_dskc[unit>>5];
     part = unit&07;
     drv = (dev &030))>>3;
     if ((partpt = dskcp->dsk_part[drv]) == NULL)
            u.u_error = ENXIO;
     else if (physck(partpt[part].nblock, B_READ))
            physio(dskstrategy, 0, dev, B_READ);
}
```

Disk read Routine Using Physical I/O

DEPENDENCIES Drivers using the **read** routine must be configured as character devices.

SEE ALSO *KPG*, "Synchronized I/O Operations"
copyin(D3X), **iomove**(D3X), **physck**(D3X), **physio**(D3X), user(D4X)

NAME

select – check whether I/O operation is possible at this time

SYNOPSIS

```
prefixselect(dev, rw)
unsigned dev;
int rw;
```

ARGUMENTS

dev	device number
rw	indicates whether this is for a read or write operation

DESCRIPTION

The **select** routine checks whether an I/O operation (type specified by the **rw** flag) issued at this time will block. If the operation would block, it returns a 0; if the operation would not block, **select** returns a 1.

The **select** routine is usually written as a **switch** statement, with separate **cases** for read and write operations. These **case** statements are coded to determine if the operation would block. For example, the code could check if the queue is empty, check the status of a device, or, for fully-semaphored drivers, check if the value of a semaphore is 0 or less.

Data Structure Used

Drivers that support **select** must initialize a driver-specific data structure (as shown in the example on page 2–54) that has:

- ❑ separate read-select and write-select members into which the proc(D4X) address of the user-level process trying to access the device is written.
- ❑ a `flags` member with separate flags to indicate that a collision occurred on a read or write operation. This flag is passed to the interrupt routine when data arrives, or when the output queue reaches the low water mark and calls **selwakeup**(D3X).

TTY Drivers

For TTY drivers that use line discipline 0, do not include code for a **select** entry point; rather, **select** functionality is provided through **ttselect**. The operating system populates `cdevsw` with **ttselect** if you configure the driver as a TTY driver with a **select** handler. Once populated, a **select**(2) call against that device calls **ttselect**, which checks whether **t_outq** is below the low water mark (for write operations) or whether there are any characters available in the canonical queue (for read operations).

RETURN VALUE

select returns a 0 (zero) if the operation would block, or a 1 (one) if the operation would not block.

DEPENDENCIES

Drivers using the **select** routine must be configured as character special devices that have a **select** handler.

Ported drivers that have a **select** routine must have the following modifications in order to work under the compatibility modes:

- ❑ The `p2_wchan` member of the **`proc`**(D4X) structure must be tested for the value **`&selwait`** to determine if a process is still attempting to execute a **select** routine on a device; on other systems, `p_wchan` is tested instead.
- ❑ If a collision occurs (two processes attempting to **select** the same device), the collision should be noted in the driver's data structures, and the driver must set the SSELCOL flag in the `proc` structure field `p_flag` (`p->p_flag |= SSELCOL`) of the process attempting to **select** the device.

SEE ALSO

selwakeup(D3X)

EXAMPLE

The next several code segments show how a driver is coded to support **select**.

The driver's header file initializes a data structure that includes read-select and write-select members and a flags member with separate flags to indicate that a collision occurred on a read or write operation, as shown below.

```
struct xxdriver_struct {
   ⋮
   struct proc *xx_rsel
   struct proc *xx_wsel
   int          xx_flags
     ⋮
}
#define XX_RCOLL 1    /* collision during read select */
#define XX_WCOLL 2   /* collision during write select */
#define XX_READABLE  4  /* device is readable */
#define XX_WRITABLE  8  /* device is writable */
```

Driver's Header File

The code that begins on page 2–56 illustrates how a **select** routine is written. Note the following:

5 This is a pointer to the device-specific data structure defined in the driver's header file. It is set up with the appropriate structure address based on the *dev* parameter.

8 The driver code that calls **selwakeup**(D3X) is usually part of the driver's interrupt routine (refer to page 2–58). If **selwakeup** is called after the driver determines that the device is not accessible for the read/write operation but before the driver's data structures have been updated to indicate that a process is attempting to select the device, the process could be blocked in the **select** code when the device is accessible. Consequently, the **selwakeup** call must be blocked until execution through this critical region has completed.

The method of preventing the **selwakeup** call varies according to the semaphoring method under which the driver is installed. For fully-semaphored drivers (as shown in the example), set a spin lock with **spsema**(D3X);[1] the spin lock must be initialized in the driver's **init**(D2X) routine.

If the driver is installed under major- or minor-device semaphoring, it is not necessary to perform any blocking action because the system locks a per-driver or per-device semaphore before entering any driver routine.

If the driver is installed under CPU affinity, an **spl**(D3X) call to block interrupts is usually sufficient.

17 - 18 Determine whether or not another process is already selecting on this device. (If so, this is a collision.) A non-zero value for `ddsp->xx_rsel` indicates that a process *may* be trying to select. We must also check that our address was not left around as stale data from a previous select attempt (line 17), and we must check that the process is really selecting (line 18). Stale data may be

[1] A kernel semaphore (set with **psema**(D3X)) can be used if the **selwakeup** call is issued only by the base level of the driver or kernel code. This is seldom done. If the device can be accessed by more than one process at a time, use the SEMRTBOOST flag with **psema**. If the device can be accessed by only one process at a time, the SEMRTBOOST flag should not be used. If the driver controls several devices or subdevices, we recommend initializing a semaphore for every device, although a global lock that blocks all data structures controlled by the driver can be used (although performance may be degraded).

left around because the process also selected on other devices that became selectable before this one.

19 – 24 If the checks described above determine that another process is already selecting on this device, a collision has occurred. Set the collision flag in the driver's data structure and in the **p_flag** member of the proc structure of the user-level process that called **select**.

26 – 41 The FWRITE case is similar to the FREAD case, except that it checks that the device is writable rather than readable, and uses different members of the driver's data structure for the device's write selects. Note that the SSELCOL flag in the proc(D4X) is set for both read and write collisions during a select operation.

```
01   xxselect(dev, rw)
02   dev_t *dev;                   /* device major/minor number */
03   int rw;                       /* read/write flag */
04   {
05   lock_t xx_drivlock;
06   struct xxdriver_struct *ddsp;

07       if (error condition exists that would be caught by read/write)
08           return(1);
09       pspsema(&xx_drivlock);

10       switch (rw) {
11       case FREAD:
12           if (ddsp->xx_flags & XX_READABLE) {
13               psvsema(&xx_drivlock);
14               return(1);
15           }
16           p = ddsp->xx_rsel;
17           if (p != 0)                /* a process has selected */
18           && (p != u.u_procp)        /* and it is not this process */
19           && (p->p_w2chan == &selwait) /* other process is selecting */
20           {
21               ddsp->xx_flags |= XX_RCOLL;
22               u.u_procp->p_flag |= SSELCOL;
23           } else {
24               ddsp->xx_rsel = u.u_procp;
25           }
26           break;
27        case FWRITE:
28             if (ddsp->xx_flags & XX_WRITABLE) {
29                  psvsema(&xx_drivlock);
30                  return(1);
31             }
32             p = ddsp->xx_wsel;
33             if (p != 0)                /* a process has selected */
34             && (p != u.u_procp)        /* and it is not this process */
35             && (p->p_w2chan == &selwait) /* other process is selecting */
36             {
37                  ddsp->xx_flags |= XX_WCOLL;
38                  u.u_procp->p_flag |= SSELCOL;
39             } else {
40                  ddsp->xx_wsel = u.u_procp;
41             }
42             break;
43        }
44        psvsema(&xx_drivlock);
45    return(0);
46    }
```

Sample select(D2X) Routine

The following code illustrates how the driver's **intr**(D2X) code is written to handle the processing for the **select** operation. Note the following:

4 The driver would set ddsp to point to the appropriate data structure, based on the value of *dev*.

6 This is the same lock used in the **select** routine.

7 – 14 If the device is writable and a process(es) is selecting for writability, **selwakeup** is invoked to unblock the process(es) and the flags are cleared to indicate no one is selecting any longer. The **select** routine will be called again from the generic system select code.

15 – 22 Similar to the above, but for read operations.

23 The **svsema** is issued after all status and flags have been updated. This allows the **select** routine to enter its critical region.

```
xxintr(dev)
dev_t dev;
{
   struct xxdriver_struct *ddsp;

      ⋮

   spsema(&xx_drivlock);

   if (the device has become writable) {
        ddsp->xx_flags |= XX_WRITABLE;
        if (ddsp->xx_wsel != NULL) {
             selwakeup(ddsp->xx_wsel, ddsp->xx_flags & XX_WCOLL);
             ddsp->xx_flags &= ~XX_WCOLL;
             ddsp->xx_wsel = NULL;
        }
   }

   if (some data has been received that can be read) {
        ddsp->xx_flags |= XX_READABLE;
        if (ddsp->xx_rsel != NULL) {
             selwakeup(ddsp->xx_rsel, ddsp->xx_flags & XX_RCOLL);
             ddsp->xx_flags &= ~XX_RCOLL;
             ddsp->xx_rsel = NULL;
        }
   }
   svsema(&xx_drivlock);
```

select Processing in the intr(D2X) Routine

NAME

serv – process a deferred interrupt

SYNOPSIS

prefix`serv(minor)`

ARGUMENTS

minor minor device number

DESCRIPTION

serv is an entry point routine that is called to service deferred interrupts for minor devices that use the minor device semaphoring feature. Interrupts for such devices are factored into two portions:

- ❑ the *prefix***intr** portion that does not need to have the driver semaphore locked
- ❑ the *prefix***serv** portion that is called only when the driver semaphore is locked

The **serv** routine is coded to handle the interrupt, as discussed on the **intr**(D2X) manual page. For drivers that are semaphored on the minor-device number, the **intr** routine is coded to defer the interrupt and call **serv** to actually handle the interrupt.

DEPENDENCIES

serv is accessed only if the driver's switch table entry is semaphored by minor device

SEE ALSO

DDG, "Porting Drivers"
intr(D2X), `semdrivs`(D4X)

NAME

strategy – handle synchronized block device input and output

SYNOPSIS

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

ARGUMENTS

bp pointer to the address of an instance of the buf(D4X) structure

DESCRIPTION

Block drivers must provide a **strategy** routine to handle the data transfer. All information to generate the job request is given in the buffer header (buf(D4X)) that is passed as the input argument. When the operation is complete, or is terminated because of an error condition, the buffer header must be updated as necessary and returned with the **iodone**(D3X) function.

strategy entry-point routines should not access the user(D4X) data structure because the process on whose behalf the transfer is to take place may not be the currently active process. Remember that some kernel functions (such as **klongjmp**(D3X), **copyin**(D3X) and **suser**(D3X)) access the user structure.

Use of buf(D4X)

All information about the data transfer is contained in the buffer header:

b_dev contains the major and minor number of the device where the I/O is to occur.

b_blkno the block number of the device where the I/O is to occur. Note that the block number is in terms of 512-byte physical blocks, not logical file system blocks.

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

b_un.b_addr the kernel physical address of the data buffer. Note that, while all kernel addresses are technically virtual addresses, much of the kernel is mapped one-to-one to physical addresses and called kernel physical memory.

b_flags the flags in the low-order 16 bits indicate the buffer status. The value of these flags should be preserved (except for B_ERROR). The high-order 16 bits are set to zero when **strategy** is called; the driver may use them in any manner. Refer to buf(D4X) for a complete list of flags; commonly used flags are:

B_READ if set, this is an input operation. If not set, this is an output operation.

B_ASYNC indicates that the transfer is taking place asynchronously, meaning that no process is blocked waiting specifically for the transfer to complete.

B_PHYS if set, this is operation will use a physical buffer

B_ERROR set by the driver in conjunction with **b_error** if the I/O operation fails

b_start can be used to time I/O operations.

The buffer header is also used to return status and error information to the kernel and the user-level program:

b_flags B_ERROR set if error occurred

b_error set to appropriate error code if error occurred

b_resid set to the number of bytes not transferred (residual byte count) if the transfer was not completed and no error was reported. This happens when the end of a transfer is not within the range of valid block numbers.

Structure of strategy Routines

The typical passage of a block device I/O operation is:

1. The **strategy** routine is called and performs initial validation checks. If validation fails, then **iodone**(D3X) is called to complete the I/O operation and **strategy** returns to the initiating process.

2. If validation is successful and the device is not busy, the operation is started immediately. If the device is busy, the operation is queued for later processing; when the device is ready to accept the request, the operation begins.

3. When the operation is complete, the device typically posts an interrupt, which is handled by the driver's **intr**(D2X) routine. **intr** checks the completion status, amends the **b_flags** and **b_error** members if an error occurred, and returns the buffer header to the caller by issuing the **iodone**(D3X) function.

The following validation checks typically are made:

- Check that the transfer count (bp->b_bcount) is for an integral number of device blocks. If not, the driver can round the transfer count down and set the **resid** member, or return the ENXIO error code.

- Check that the given block number is valid. If not, return ENXIO.

- Check that the given block number (expressed in terms of 512-byte physical blocks) maps correctly to the device's block size. For instance, if the device uses 1-Kbyte blocks (each device block contains two physical blocks), the given block number must be a multiple of 2; if the device uses 2-Kbyte blocks (each logical block contains four physical blocks), the given block number must be a multiple of 4. If the block number does not map to the device's block size correctly, return ENXIO.

- Check that the device is operational if necessary; usually this is done in the **open**(D2X) routine.

- Check if the transfer would start at or past the end of the partition.

  - If the transfer is exactly at the end and a read operation is required, set the residual byte count (**b_resid**) and call **iodone**(D3X).

  - If it would start within partition bounds but go beyond it, set **b_resid** for the amount not transferred and set up the read/write operation for the portion of the transfer that is allowed.

When validation tests in the **strategy** routine fail, the driver:

- sets the B_ERROR flag in **b_flags** (unless **b_resid** was set)

- writes an appropriate error code (usually ENXIO) to **b_error** (unless **b_resid** was set)

- calls the **iodone**(D3X) routine to terminate the operation. If a user-level process is awaiting the results of the **strategy** routine, the kernel propagates any error code in **b_error** via **u.u_error** to a system call error return to the calling process.

The following code fragment illustrates this:

```
if (dp->b_bcount & (BSIZE-1)) {
     bp->b_flags |= B_ERROR;
     bp->b_error = ENXIO;
     iodone(bp);
     return;
}
```

The driver should be written so that **strategy** calls do not fail because of resource constraints. If, for example, each **strategy** call requires an instance of a control block, of which only a limited number are available, it must block on a semaphore until the resource becomes available. This waiting is undesirable; the driver should be configured so it is guaranteed to have sufficient resources for the maximum possible number of outstanding strategy calls. This maximum number can be calculated by adding:

- □ the number of buffers in the system buffer cache (viewable as the **v_buf** field on the **var** output of **crash**(1M); this shows the total number of buffers of all sizes)

- □ the number of buffers in the physical buffer cache (viewable as the **v_pbuf** field in on the **var** output of **crash**)[1]

For example, if **v_var** is 760 and **v_pbuf** is set to 50, the maximum number of simultaneous **strategy** routines that could be executing is 810.

If the buffer header is to be entered into a queue, the typical practice is to use the **av_forw** and **av_back** pointers to enter it into a doubly-linked list. Care should always be taken to ensure that any list manipulation be protected. Use **spsema**(D3X) to set a spin lock before executing the list manipulation code[2], and **svsema**(D3X) to unlock the spin lock after queuing has been performed. Other queuing methods are allowed.

[1] The number of buffers in the system buffer cache and the physical buffer cache are determined by tunable parameters. Refer to the *System Administrator's Guide* for more information.

[2] Drivers installed under CPU affinity use the **spl***(D3X) functions to disable interrupts before sending the request and **splx_fast** to reenable interrupts after the request is sent to the controller. For drivers installed under major- or minor-device semaphoring, the operating system protects the code section from interrupts; **spl*** functions are legal, but will unnecessarily impair the interrupt latency of the system. Note that not all machines support CPU affinity; refer to the Release Notes shipped with your system.

strategy Routines in Character Drivers

In block drivers that also support character access, the **read**(D2X) and **write**(D2X) routines (accessed through cdevsw(D4X)) may call **strategy** as a subordinate routine. In this case, if **b_un.b_addr** is a user virtual address, the **strategy** routine may examine the **u.u_segflg** member of user(D4X) to determine the type of address passed in **b_un.b_addr**.

The B_PHYS flag must always be set when **strategy** is called as a subordinate routine for character access, to indicate that the transfer is not going to the kernel buffer cache. (The buf(D4X) header is used to control the transfer, but is not associated with an actual kernel buffer). The buffer size given in **b_bcount** may differ from the normal buffer size, and the address in **b_un.b_addr** may not be a kernel address.

If **b_un.b_addr** refers to an area of user virtual memory, then an additional member of buf can be used:

b_proc contains a pointer to the proc(D4X) structure that **strategy** can use to perform a mapping of user address space to physical addresses.

This mapping of user address space to physical addresses is not used in any existing REAL/IX drivers, and customers who use it must take care to ensure that the area is locked down through **userdma**(D3X) or some similar function.

DEPENDENCIES

Drivers using the **strategy** routine must be configured as block devices. If the driver also supports character access, it must also be configured as a character device.

SEE ALSO

KPG, "Synchronized I/O Operations"
intr(D2X), **mbstrategy**(D2X), **print**(D2X), **physio**(D3X), buf(D4X)

NAME

write – write data to a character-access device (synchronous I/O)

SYNOPSIS

```
prefixwrite(dev)
dev_t dev;
```

ARGUMENTS

dev	device number

The following members of the user(D4X) structure are implicit arguments to the **write** routine:

u.u_base address of the buffer in user virtual memory where the **write** data is to be found

u.u_count byte count for the data transfer

u.u_ap points to the original parameters of the **write**(2) system call

u.u_segflg set to 0

u.u_fmode copy of the **f_flag** member of the file structure (defined in *sys/file.h*). The flag propagates the modes set in the **open**(2) request.

u.u_offset current offset in the file

DESCRIPTION

When **write** 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.

The **write** routine typically does the following:

- ❑ validate device number; if invalid, set **u.u_error to** ENODEV
- ❑ Initiate the data transfer:
  - For TTY drivers, use the **ttwrite**(D3X) function to do the transfer using the tty(D4X) structure to get a cblock(D4X) for buffering the transfer and update the user(D4X) structure. This is generally used for low-speed character devices.
  - For raw I/O on a block device, use the **physck**(D3X) and **physio**(D3X) functions to initiate the transfer. **physio** handles memory page locking to ensure that the pages impacted by the I/O are

not swapped out and does the unbuffered I/O while maintaining the buffer header as the interface structure.

- For other character drivers, use the **copyout**(D3X) function to move the data from the user area to the kernel buffer area and from the kernel buffer area to the device. If not using one of the system-supplied buffering schemes, the driver must set up its own buffering scheme; this is generally used with high-speed character devices such as network interface boards.

❑ Block on a semaphore with **psema**(D3X) to suspend execution until the I/O operation is complete. (If the driver entry in cdevsw is semaphored, you can suspend execution with **sleep**(D3X).)

❑ After the **intr**(D2X) routine unblocks the semaphore with a **vsema** (or **wakeup** for drivers that blocked with **sleep**) signaling that the I/O operation is complete, return back to the associated user-level process.

Return Values

On return from the driver, the following members of the user(D4X) structure are used to generate the return values for the **write**(2) system call:

u.u_error set if an error occurred during the I/O operation

u.u_count set to the residual byte count (in other words, the amount (if any) of the requested transfer that could not be transferred. Set to 0 if all data was transferred.

In addition, the byte count parameter supplied by the user (pointed to, along with other parameters, by the **u.u_ap** member) may have been changed. The **write**(2) system call calculates the number of bytes transferred as the difference between the byte count parameter and the residual byte count in **u.u_count**. If, for example, the write is going to a block device and would extend beyond the limits of the device, the driver may scale down the request before passing it to a **strategy**(D2X) routine. There is no residual byte count from the scaled down request, but the transfer count returned from the system call has to reflect the reduced transfer size. This can be achieved by setting the byte count parameter to the lower value.

write Routines that use physio(D3X)

Refer to **read**(D2X) for a discussion of **read** routines that use physical I/O. A sample **write**(D2X) routine that uses **physio**(D3X) is:

```
dskwrite(dev)
register dev_t dev;
{
     register unit                     /* disk controller ID */
     register unsigned char drv;       /* disk drive ID */
     register struct dskc *dskcp;      /* disk controller pointer */
     register struct dskpart *partpt;  /* pointer to partition info */
     register unsigned char part;      /* drive partition */

     unit = minor(dev);
     dskcp = &dsk_dskc[unit>>5];
     part = unit & 07;
     drv = (dev & 030)>>3;
     if ((partpt = dskcp->dsk_part[drv]) == NULL)
          u.u_error = ENXIO;
     else if (physck(partpt[part].nblock, B_WRITE))
          physio(dskstrategy, 0, dev, B_WRITE);
}
```

Disk write(D2X) Routine Using Physical I/O

DEPENDENCIES Drivers using the **write** routine must be configured as character devices.

SEE ALSO *KPG*, "Synchronized I/O Operations"
aio(D2X), **read**(D2X), **copyout**(D3X), **iomove**(D3X), **physck**(D3X), **physio**(D3X)

Chapter 3

Kernel Functions and Macros (D3X)

Section D3X describes the driver functions and macros that serve as library functions for device drivers.[1] The functions are presented on separate pages. All manual pages for kernel functions and macros have the (D3X) cross reference code.

Section D3X includes information about macros that we anticipate our customers will need. Macros are defined in header files in the */usr/include/sys* directory, and kernel programmers can look through those files to locate other macros that may be required. Note especially a number of memory conversion macros in *immu.h* and general macros in *sysmacros.h*.

Manual pages in this section contain the following headings:

NAME summarizes the function's purpose

SYNOPSIS describes the function's entry point in the source code. Note that the **#include** lines listed for each function generally do not include the header files that are required for every driver; refer to the *Kernel Programming Guide* for information about these standard header files. Typically, kernel-level code should include, at a minimum, the following lines (in the order given):

```
#include <sys/inline.h>
#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/param.h>
#include <sys/errno.h>
```

ARGUMENTS describes any arguments required to invoke the function

DESCRIPTION describes general information about the function

[1] Some functions and macros described in this section may not be supported on your machine. Refer to the Release Notes shipped with your system.

SEMAPHORE RAMIFICATIONS
explains whether or not spin locks and semaphores can be held when calling the function, and identifies functions that can be used only in a fully-semaphored driver or only in a driver installed under one of the compatibility modes[1]

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

SOURCE FILE indicates the file name where the function or macro is defined. Kernel source files are located in the */usr/src/uts/realix* directory.

SEE ALSO indicates functions that are related by usage and lists sources of additional information. The following abbreviations are used:

KPG for the *Kernel Programming Guide*
DDG for the *Driver Development Guide*

EXAMPLE provides an expansion of the information in a usable context

Function Categories

Table 3–1 groups the kernel functions by category. Refer to individual manual pages in this section for details about each function.

VMEbus

In Table 3–1, the following kernel functions can be used only on VMEbus-based systems: **usshmctl**, **vme_a24_mem_valid**, **usyscall**.

386

In Table 3–1, the following kernel functions can be used only on 386/486-based systems: **inb**, **inw**, **inl**, **outb**, **outw**, **outl**, **io_alloc**, **debug**.

In addition to the categories listed in Table 3–1, two functions – **nodev** and **nulldev** – are provided for informational purposes, but are not used directly in a driver.

[1] Not all compatibility modes are supported on all machines. Refer to the Release Notes shipped with your system.

Table 3–1. Function Categories

Category	Functionality	Kernel Function Name
Kernel Semaphores	Initialize a semaphore	**initsema**
	Lock (decrement) a semaphore	**psema, cpsema**
	Unlock (increment) a semaphore	**vsema, cvsema**
	Check the value of a semaphore	**valusema**
	Decrement a semaphore value for a resource	**decsema**
	Increment a semaphore value for a resource	**incsema**
Spin Locks	Initialize a spin lock	**initlock**
	Set a spin lock	**spsema**
	Release a spin lock	**svsema**
	Check the value of a spin lock	**valulock**
Timing Functions	System calls and semaphored drivers	**delayfs, timeoutfs, timeoutfspri, untimeout,**
	Driver compatibility modes	**delay, timeout, timeoutpri, untimeout**
	Delay by spinning independent of clock	**DELAY**
	Get, set, and release interval timer	**get_timer, set_timer, rel_timer**
Synchronization for Driver Compatibility Modes	Block and unblock a process	**sleep, wakeup**
	Prevent/allow interrupts	**spl*, splx, splx_fast, disable, enable, popsr, pushsrdisable**
Connected Interrupts	Connect the driver to a `cintrio`(4) structure	**cintrget**
	Implement connected interrupt IOCTLs	**cintrctl**
	Notify the associated user-level process of a device interrupt	**cintrnotify**
	Release the `cintrio`(4) structure	**cintrelse**
Asynchronous I/O	Register completion of the I/O operation	**comp_aio**
	Register cancellation of the I/O operation	**comp_cancel_aio**

Table 3–1. Function Categories (cont.)

Category	Functionality	Kernel Function Name
Data Movement	Copy data from a driver to a user program	**copyout, subyte, suword, iomove**
	Copy data from a user program to a driver	**copyin, fubyte, fuword, iomove, upath**
	Copy data in kernel space	**bcopy**
Block I/O	Allocate and deallocate buffers	**geteblk, getnblk, brelse**
	Clear a buffer	**clrbuf**
	Suspend when I/O begins	**iowait, preiowait**
	Report when I/O transfer completes	**iodone**
	Read and write raw data for a block device	**physck, physio, dma_breakup**
Character I/O	Read data	**getc, getcb, getcf, cpass, inb**[a]**, inw**[a]**, inl**[a]
	Write data	**putc, putcb, putcf, passc, outb**[a]**, outw**[a]**, outl**[a]
Strings	Compare strings	**strcmp, strncmp**
	Copy one string to another	**strcpy, strncpy**
	Obtain number of characters in a string	**strlen**
TTY Subsystem	Clear buffer	**ttyflush**
	Delay a process	**tttimeo, ttywait, ttrstrt**
	I/O control	**ttiocom, ttioctl**
	Open/close terminal	**ttopen, ttinit, ttclose**
	Read from a terminal	**canon, ttin, ttread**
	Write to a terminal	**ttout, ttwrite, ttxput**

Table 3–1. Function Categories (cont.)

Category	Functionality	Kernel Function Name
Memory Management	Allocate and deallocate memory	**bmemalloc, bmemfree, sptalloc, sptfree, freecpages, freephysbuf, getcpages, getphysbuf**
	Lock and map user virtual memory to kernel virtual memory	**kmap**
	Unmap and unlock user virtual memory from kernel virtual memory	**kunmap**
	Lock and unlock user virtual memory for direct memory access	**userdma, undma**
	Clear memory	**bzero**
	Obtain real addresses of pages in user buffer	**disjointio**
	Obtain page physical address	**pg_getaddr**
	Obtain page number, offset, number within a segment	**pnum, poff, pshum**
	Obtain segment number, offset	**snum, soff**
	Obtain page descriptor entry for user virtual address	**uvtopde**
	Manage a private buffer scheme	**malloc, mapinit, mfree**
	User-defined special shared memory	**usshmctl**[b]
	Allocate memory-mapped IO address space	**io_alloc**[a]
	Probe to determine if a device is present	**bprobe, lprobe, sprobe**
	Flush virtual data cache	**dcachclr**
	Verify A24 address	**vme_a24_mem_valid**[b]
Data Structures	Allocate and deallocate disjoint I/O structure	**djntget, djntfree**
	Allocate and deallocate physical I/O buffer header	**getpbp, freepbp**
	Prevent compiler from reporting unaligned structures in kernel	**NOT_ALIGNED**
User-Defined Functions	Action(s) to take after a system panic	**atpanic**
	Action(s) to take after AC power fails	**atpfail**

Table 3–1. Function Categories (cont.)

Category	Functionality	Kernel Function Name
Miscellaneous	Lock and unlock semaphore on bdevsw or cdevsw	**drilock, driunlock, driinvoke**
	Compare integers	**max, min**
	Convert between bytes and clicks	**btoc, ctob**
	Display message or panic the system	**cmn_err**
	Access device number	**major, minor, makedev**
	Non-local goto, typically used to return control to user program with error code set	**klongjmp, olongjmp, ksetjmp, osetjmp**
	Signal user-level process(es)	**psignal, psignalcur, psignalval, signal, send_event**
	Verify user access	**rtuser, suser, useracc**
	Debugging	**debug**[a]
	Add a function name to and remove a function name from a list of functions to be executed when the process exits or **execs**	**ee_add, ee_rm**
	Unblock process waiting to select a device	**selwakeup**
	Install user-defined system call	**usyscall**[b]

[a] Applicable only on a 386/486-based system.

[b] Applicable only on a VMEbus-based system.

Summary of Kernel Functions

Table 3–2 lists the kernel functions and their descriptions in alphabetical order. The following conventions are used in the "Type" column:

B	Used only in block drivers
C	Used only in character drivers
G	Generic (used in block and character drivers)
i	Can be called from an interrupt routine
s	Can be called from the **strategy** routine
E	Only for compatibility-mode driver
F	Only for fully-semaphored driver
P	Can be used with either fully-semaphored or compatibility-mode driver
T	Semaphoring must match TTY subsystem

VMEbus

In Table 3–2, the following kernel functions can be used only on VMEbus-based systems: **usshmctl, usyscall, vme_a24_mem_valid.**

386

In Table 3–2, the following kernel functions can be used only on 386/486-based systems: **debug, inb, inl, inw, io_alloc, outb, outl, outw.**

Table 3–2. Kernel Function Summary

Routine	Description	Type
atpanic( )	system function called when system panics	P
atpfail( )	system function called when AC power fails	P
bcopy(*from, to, bcount*)	copies data between locations in the kernel; for example, from one buffer to another	GisP
bmemalloc(*siz*)	allocates *siz* number of bytes of memory	GsP
bmemfree(*vaddr, siz*)	frees memory allocated with **bmemalloc**	GsP
bprobe(*addr, val*)	tests for the presence of a device (byte address)	GsP
brelse(*bp*)	returns buffer to the kernel	BisP
btoc(*bytes*) **btoct**(*bytes*)	returns the number of clicks (swappable memory pages) in the specified number of bytes	GisP
bzero(*addr, bytes*)	clears memory for a number of bytes	GisP
canon(*tp*)	performs canonical processing	CET
cintrctl(*cid, command, arg*)	implements connected interrupt IOCTLs	CP
cintrelse(*cid*)	releases a cintrio structure	CP
cintrget(*key, arg, flag*)	connects driver to a cintrio structure	CP
cintrnotify(*cid, dataitem*)	notifies user-level process of interrupt	CiP
clrbuf(*bp*)	erases buffer contents	BisP
cmn_err(*level, format, args*)	displays message	GisP
comp_aio(*areq, byte_cnt, status*)	marks completion of asynchronous I/O	CiF
comp_cancel_aio(*areq*)	marks cancellation of asynchronous I/O	CiF
copyin(*userbuf, driverbuf, count*)	copies data from user space to the driver	GP
copyout(*driverbuf, userbuf, count*)	copies data from the driver to user space	GP
cpass( )	gets next character from user's write call	CP
cpsema(*sem_addr, flags*)	locks semaphore for a resource only if resource is available	GisF
ctob(*clicks*)	returns the number of bytes in the specified number of clicks (swappable memory pages)	GisP
cvsema(*sem_addr*)	unlocks semaphore (makes resource available) if a process is waiting	GisF
dcachclr( )	clears virtual data cache	GiP
debug( )[a]	invokes the kernel debugger	GisP
decsema(*sem_addr*)	decrements semaphore by 1 (statistics only)	GisF
DELAY(*microseconds*)	delays by spinning independent of system clock	GiP
delay(*ticks*)	delays for *ticks* clock ticks	GsE
delayfs(*ticks*)	delays for *ticks* clock ticks	GsF
disable( )	disables interrupts for the processor	GP
disjointio(*bp,djntprtr,szdjnt,maxtc*)	gets physical location of user virtual memory	GP
djntfree(*entryp*)	frees a disjoint I/O structure	GiP
djntget(*slpflg*)	allocates a disjoint I/O structure	GP

Table 3–2. Kernel Function Summary (cont.)

Routine	Description	Type
dma_breakup(*strat, bp, sectorsize*)	sets up intermediate kernel buffering for **physio**	CsP
driinvoke(*sw, maj, min, rtne, parm*)	fast locks on switch tables for driver semaphoring	GF
drilock(*switch, major, minor*)	locks a switch table entry	GsF
driunlock(*switch, major, minor*)	unlocks a switch table entry	GsF
ee_add(*func*)	adds a function name to a list of functions	GP
ee_rm(*func*)	removes a function name from a list of functions	GP
enable( )	reenables all interrupts	GiP
freecpages(*paddr, npgs*)	frees contiguous pages allocated with **getcpages**	GiP
freepbp(*bp*)	frees buffer header obtained with **getpbp**	CisP
freephysbuf(*buffp*)	releases physical buffer obtained with **getphysbuf**	CisP
fubyte(*userbuf*)	copies a byte from user to driver	GP
fuword(*userbuf*)	copies a word from user to driver	GP
getc(*clp*)	gets first byte from clist	CiET
getcb(*clp*)	gets first cblock on clist	CiET
getcf( )	gets a free cblock	CiET
getcpages(*npgs, mode*)	gets physically contiguous pages	GiP
geteblk( )	gets an empty buffer	GsP
getnblk(*bf, need*)	gets an empty buffer of specified size	GsP
getpbp(*slpflg*)	gets physical I/O buffer pointer	GisP
getphysbuf(*size*)	gets physical buffer	GsP
get_timer(*type*)	gets an interval timer	GisP
inb(*port*)[a]	read an 8-bit value (byte) at 80x86 I/O address (port)	GisP
inw(*port*)[a]	read a 16-bit value (short) at 80x86 I/O address (port)	GisP
inl(*port*)[a]	read a 32-bit value (long) at 80x86 I/O address (port)	GisP
incsema	increments a semaphore	GisF
initlock(*lock_addr, lock_val, flags*)	initializes spin lock	GF
initsema(*sem_addr, sem_val, flags*) **reinitsema**(*sem_addr,sem_val, flags*)	initializes or reinitializes semaphore for a resource	GF GiF
io_alloc( )[a]	allocate memory mapped virtual address space	GisP
iodone(*bp*)	signals completion of I/O after **iowait**	BisP
iomove(*cp, bytes, rwflag*)	moves *bytes*	CP
iowait(*bp*)	blocks execution to wait for block I/O to complete	GP
klongjmp( )	jumps back to location of **u.u_qsav**	GsP
kmap(*base, count*)	locks user virtual memory and maps it to kernel virtual memory	GP
ksetjmp( )	saves registers and return location for **ksetjmp**	GP

Table 3–2. Kernel Function Summary (cont.)

Routine	Description	Type
kunmap(*base, count, kvaddr*)	unmaps and unlocks user virtual memory from kernel virtual memory	GP
lprobe(*addr, val*)	tests for the presence of a device (32-bit address)	GsP
major(*dev*)	returns major number from device number	GisP
makedev(*majnum, minnum*)	creates a device number	GisP
malloc(*mp, size, waitflg*)	allocates space from a map structure	GsP
mapinit(*map, mapsize, s1, s2*)	initializes map structure	GisP
max(*int1, int2*)	returns the larger integer	GisP
mfree(*mp, size, a*)	returns space to a map structure	GisP
min(*int1, int2*)	returns the smaller integer	GisP
minor(*dev*)	returns minor number from device number	GisP
nodev( )	returns an error upon access	See Note
NOT_ALIGNED	specifies that compiler does not complain about structure that is not aligned	GP
nulldev( )	performs no operation	See Note
olongjmp(*save_area*)	jumps back to location saved by **osetjmp**	GsP
osetjmp(*save_area*)	saves registers and return location for **olongjmp**	GP
outb(*port, value*)[a]	write an 8-bit value (byte) to I/O address (port)	GisP
outw(*port, value*)[a]	write a 16-bit value (short) to I/O address (port)	GisP
outl(*port, value*)[a]	write a 32-bit value (long) to I/O address (port)	GisP
passc(*c*)	passes character to user-level process	CP
pg_getaddr(*p*)	gets page address	GiP
physck(*nblocks, rwflag*)	verifies block exists	GsP
physio(*strat, bp, dev, rwflag*)	calls **strategy** routine for direct block I/O	GsP
pnum(*addr*)	gets page number	GiP
poff(*addr*)	gets page offset	GiP
popsr( )	enable interrupts and restore saved interrupt privilege level (ipl)	GisP
preiowait(*bp*)	blocks execution to wait for block I/O to complete	GsP
psema(*sem_addr, flags*)	locks semaphore for a resource	GiF
psignal(*p, signal*)	sends signal to a process	GiP
psignalcur(*p, sigmask*)	sends signal to currently executing process	GiP
psignalval(*p, signum, sigmask*)	sends signal to specified process	GiP
psnum(*addr*)	gets page number within segment	GiP
pushsrdisable( )	disable interrupts and save current interrupt privilege level (ipl)	GisP
putc(*c, clp*)	puts byte on clist	CiET
putcb(*cbp, clp*)	links a cblock to the clist	GisET
putcf(*cbp*)	puts cblock on free list	GiET

Table 3–2. Kernel Function Summary (cont.)

Routine	Description	Type
rel_timer(*tp*)	releases an interval timer obtained with **get_timer**	GisP
rtuser( )	verifies realtime permission mode	GsP
selwakeup(*proc, coll*)	notifies base level that device is selectable	GiP
send_event(*p, eid, type, ditem*)	posts an event to a user process	GisP
set_timer(*tp, type, val, oval, func, funcarg*)	sets an interval timer obtained with **get_timer**	GsP
signal(*pgrp, signal*)	sends signal to process group	GisP
sleep(*addr, priority*)	suspends execution	GsE
snum(*addr*)	gets segment number	GiP
soff(*addr*)	gets segment offset	GiP
spl*( )	suspends or allows interrupts	GisP
splx(*oldlevel*) or **splx_fast**(*oldlevel*)	restores *oldlevel* of interrupts	GisP
sprobe(*addr, val*)	tests for the presence of a device (16-bit address)	GsP
spsema(*lock_addr*)	sets a spin lock	GisF
sptalloc(*size, mode, base*)	allocates memory pages	GP
sptfree(*vaddr, size, mode*)	frees allocated memory pages	GP
strcmp(*s1, s2*) **strncmp**(*s1, s2, n*)	compares strings	GiP
strcpy(*s1, s2*) **strncpy**(*s1, s2, n*)	copies string *s2* to *s1*	GiP
strlen(*s*)	returns length of specified string	GiP
subyte(*userbuf, c*)	copies a byte from driver to user	GP
suser( )	verifies superuser permission mode	GsP
suword(*userbuf, i*)	copies a word from driver to user	GsP
svsema(*lock_addr*)	releases a spin lock	GisF
timeout(*func, arg, ticks*)	calls function in *ticks* clock ticks	GiE
timeoutfs(*func, arg, ticks*)	calls function in *ticks* clock ticks	GiF
timeoutfspri(*func, arg, ticks*)	same as **timeoutfs** except allows the operating system to arrange for daemon of appropriate priority level to handle timeout processing	GF
timeoutpri(*func, arg, ticks*)	same as **timeout** except allows the operating system to arrange for daemon of appropriate priority level to handle timeout processing	GE
ttclose(*tp*)	closes a TTY device	CET
ttin(*tp, code*)	moves character(s) to raw queue	CiET
ttinit(*tp*)	opens a closed TTY device; initializes tty structure with default setting on an initial open	CiET
ttiocom(*tp, cmd, arg, mode*)	changes device parameters	CET
ttioctl(*tp, cmd, arg, mode*)	sets device parameters	CET
ttopen(*tp*)	opens a TTY device	CET

Table 3–2. Kernel Function Summary (cont.)

Routine	Description	Type
ttout(*tp*)	moves a TTY character from user data space to an output queue	CiET
ttread(*tp*)	moves TTY characters from canonical queue to user	CET
ttrstrt(*tp*)	restarts TTY output	CiET
tttimeo(*tp*)	times terminal read request	CiET
ttwrite(*tp*)	moves TTY byte from output queue to transmit buffer	CET
ttxput(*t, ucp, ncode*)	puts data in TTY output buffer	CisET
ttyflush(*tp, rwflag*)	clears a cblock and wakens processes sleeping on completion of I/O	CiET
ttywait(*tp*)	suspends TTY processing until I/O completes	CsET
undma(*base, count, rw*)	unlocks memory locked with **userdma**	GP
untimeout(*id*)	cancels **timeout** or **timeoutfs** with matching ID	GisP
upath(*userbuf, kernelbuf, maxbufsz*)	copies data from user space to kernel space	GsP
useracc(*base, count, access*)	verifies user access to data structures	GsP
userdma(*base, count, rw*)	locks user virtual memory for DMA transfer	GP
usshmctl(*sshmtype, func*)[b]	installs a user-defined special shared memory control function into the kernel	GP
usyscall(*nsyscall, func, nargs*)[b]	installs user-defined system call into kernel	GP
uvtopde(*uva*)	returns page descriptor entry for user virtual address	GsP
valulock(*lock_addr*)	returns current value of the spin lock	GisF
valusema(*sem_addr*)	returns current value of the semaphore	GisF
vme_a24_mem_valid(*paddr, bufsiz*)[b]	verifies that an address is accessible by A24 VME devices	GisP
vsema(*sem_addr, reserved, flags*)	unlocks a semaphore, unblocks process if waiting	GisF
wakeup(*addr*)	resumes blocked execution	GisP

Note: This function is not called from a driver.

[a] Applicable only on a 386/486-based system.

[b] Applicable only on a VMEbus-based system.

Portability Issues

When discussing kernel-level portability, it is important to remember that there is no standard on kernel code: neither SVID nor POSIX addresses anything below the system-call level, and all that is standardized for system calls is a basic set to be included, not the lower-level kernel functions used to implement the system calls. Consequently, each kernel has a number of variations from other kernels. In addition to modifications made to provide performance that is acceptable for realtime applications, the REAL/IX Operating System includes some modifications to the UNIX System V kernel made when the operating system was ported to the microprocessor unit on which your machine is based.

As a starting point, the tables on the following pages compare the REAL/IX kernel to that documented in the AT&T UNIX System V Release 3 *Driver Reference Manual*. If the kernel code you are porting ran on a different variation of the operating system, you may find additional inconsistencies. At worst, these changes should be a minor aggravation. If you have code to port, a simple **grep**(1) should enable you to identify all UNIX System V entry-point routines and kernel functions that are not supported. To identify other variations, you can carefully compare the code to the routines and functions listed in this section, or you can attempt to compile the driver code; the linker will flag functions that are not supported as unresolved references.

AT&T documents a number of kernel functions that are not supported on the REAL/IX Operating System. Some of these are machine-specific functions that are not included in the porting base; some are not included in the system from which the REAL/IX Operating System was ported; others were changed because of specific issues related to the REAL/IX Operating System.

Table 3-3 summarizes the kernel functions documented in the *AT&T Driver Reference Manual* that either are not supported or are used differently on the REAL/IX Operating System, with guidelines on how to modify code that calls these functions.

The D3X kernel functions listed in Table 3-4 are implemented only on the REAL/IX Operating System. Sections of code that use these functions should be considered non-portable and should be isolated appropriately. Note that the system from which the REAL/IX Operating System was ported also includes a number of kernel functions that were not documented by AT&T; these functions are not listed in Table 3-4 but are documented in this section.

Table 3–3. AT&T Kernel Functions Not Supported

AT&T UNIX System V, Release 3	REAL/IX System Release C.0
delay(*ticks***)**	No change if installed under compatibility mode.
	Replace with **delayfs** if driver code is fully semaphored.
drv_rfile(*D_FILE***)**	Not supported
hdeeqd(*dev, pdsno, edtyp***)**	Not supported; SCSI disk devices have own hard-disk error reporting scheme implemented
hdelog(*eptr***)**	
iowait(*bp***)**	While still supported, virtually all driver calls to this function should be replaced with **preiowait**(D3X). Refer to the **preiowait** reference page for more information.

kseg(*pages***)**
unkseg(*vaddr***)**
	Not supported; to allocate/deallocate memory pages from a map, use **sptalloc** and **sptfree**.

logmsg(*message***)**	Not supported
longjmp(*env***)**	If *env* is **u.u_qsav**, use **klongjmp** with no argument; for all other values of *env*, use **olongjmp**
malloc(*mp, size***)**	semantics are changed; refer to manual page for details
mapinit(*map, mapsize***)**	semantics are changed; refer to manual page for details
mapwant(*vaddr***)**	In fully semaphored drivers, **mapwant** is called automatically.

sleep(*event, priority***)**
wakeup(*event***)**
	Can be used only if driver entry is semaphored; *priority* argument has slightly different meaning.

	For fully semaphored drivers, replace with **psema** and **vsema**.
spl*( )	Can be used as-is with drivers installed under CPU affinity,[a] although note that the spl-to-IPL relationship is usually different for each computer. For increased performance, replace calls to **splx** with calls to **splx_fast**.
	Can be removed from drivers installed under major or minor device semaphoring to improve interrupt latency on system, except when it protects a resource that is shared with other kernel processes.
	For drivers that are fully semaphored, most **spl**s can be replaced with spin locks (**spsema** and **svsema**)
sptalloc(*size, mode, base, flag***)**	Semantics are changed; refer to **sptalloc**(D3X) for details.
sptfree(*vaddr, size, mode***)**	Semantics are changed; refer to **sptfree**(D3X) for details.
timeout(*func, arg, ticks***)**	No change if driver is installed under a compatibility mode.
	Replace with **timeoutfs** if driver code is fully semaphored.
vtop(*vaddr, p***)**	Not supported

[a] Not all machines support CPU affinity. Refer to the Release Notes shipped with your system.

Table 3–4. REAL/IX–Only Kernel Functions

Feature	D3X Function	Description
Connected Interrupts	**cintrctl(***cid, command, arg***)**	Implement connected interrupt IOCTLs
	cintrelse(*cid***)**	Release a connected interrupt identifier
	cintrget(*key, arg, flag***)**	Connect driver to a connected interrupt structure
	cintrnotify(*cid, dataitem***)**	Notify user-level process of device interrupt
Kernel Semaphores (Suspend Locks)	**initsema(***sem_addr, sem_val, flags***)**	Initialize kernel semaphore
	psema(*sem_addr, flags***)**	Decrement semaphore; block if unavailable
	cpsema(*sem_addr, flags***)**	Decrement semaphore; return if unavailable
	vsema(*sem_addr, proc, flags***)**	Increment semaphore
	valusema(*sem_addr***)**	Return current value of semaphore
	preiowait(*bp***)**	Wait for completion of block I/O
Spin Locks	**initlock(***lock_addr, lock_val***)**	Initialize spin lock
	spsema(*lock_addr***)**	Lock spin lock
	svsema(*lock_addr***)**	Unlock spin lock
	valulock(*lock_addr***)**	Return current value of spin lock
Performance	**klongjmp( )**	Replaces **longjmp**
	splx_fast(*x***)**	A faster alternative to **splx**
Kernel Semaphores	**drilock(***switch, major, minor***)**	Lock a switch table entry
	driunlock(*switch, major, minor***)**	Unlock a driver entry
Asynchronous I/O	**comp_aio(***areq, byte_cnt, status)*	Mark completion of asynchronous I/O operations
	comp_cancel_aio(*areq***)**	Cancel asynchronous I/O operation
Realtime Signals	**psignalcur(***pid, sigmask***)**	Signal currently executing process
	psignalval(*pid, sigmask***)**	Signal specified process
	send_event(*pid, eid, type, dataitem***)**	Post event to specified process
Memory Management	**bmemalloc(***siz***)**	Allocate *siz* number of bytes of memory
	bmemfree(*vaddr, siz***)**	Free memory allocated with **bmemalloc**
Panic and Powerfail Handling	**atpanic( )**	Function to execute after a system panic
	atpfail( )	Function to execute after an AC power failure

Table 3-4. REAL/IX-Only Kernel Functions (cont.)

Feature	D3X Function	Description

Other	**ksetjmp(***addr***)**
	klongjmp( )
	osetjmp(*addr***)**
	olongjmp( )
		Provides **longjmp** functionality in the semaphored kernel

NAME

atpanic – function to execute after a system panic

SYNOPSIS

```
atpanic()
```

ARGUMENTS

None.

DESCRIPTION

The **atpanic** function is called when the system panics. The released system includes an **atpanic** function that does nothing but return 1 to let the panic proceed; you can define your own **atpanic** function by putting the code in the *custom.c* file specified below.

Each executing kernel can have only one **atpanic** function, so the function must be defined to handle all situations needed by any kernel program. Note that there is no guarantee that the system will be able to call **atpanic**, and that code that stops a potential panic can be very dangerous if not thought out and implemented carefully.

SEMAPHORE RAMIFICATIONS

Because it is impossible to predict what will be executing at the time the panic occurs, the **atpanic** function must be coded to have no semaphore ramifications.

RETURN VALUE

As released, **atpanic** returns 1 under all conditions. Return codes have the following meaning to **atpanic**:

0	stop the panic
1	let the panic proceed

Your **atpanic** function can include code for both return values. If you stop the panic (return 0), the panic error message is not displayed.

LEVEL

Base or Interrupt

SOURCE FILE

/stub/atpanic.c (code should be put in *usr/src/uts/realix/custom/atpanic.c*)

SEE ALSO

atpfail(D3X)

EXAMPLE

A simple example of coding in **atpanic** is the following, which writes a message to the console and **putbuf**, then lets the panic proceed:

```
cmn_err(CE_NOTE, "My atpanic handler has been invoked.");
return 1;
```

NAME

atpfail – function to execute when system suffers an AC power failure

SYNOPSIS

`atpfail()`

ARGUMENTS

None.

DESCRIPTION

The **atpfail** function is called when the system suffers a power failure. It executes in the few microseconds between the power failure and when the system actually runs out of power.

The released system includes an **atpfail** function that does nothing but return 1. You can define your own **atpfail** function by putting the code in the *custom/atpfail.c* file specified below. The primary reason for defining your own function is to ensure it takes whatever action is locally suitable for a system that is going down very shortly. It is important to note that if you define your own **atpfail** function, it must not call any routines that may wait on a semaphore or spin lock; in particular, it should not call any kernel routines.

Each executing kernel can have only one **atpfail** function, so the function must be defined to handle all situations needed by any kernel program. Note that there is no guarantee that the system will be able to call **atpfail.** If the system is configured with an uninterruptible power supply (UPS), it may not even realize that it has suffered a power failure to call this routine.

SEMAPHORE RAMIFICATIONS

Because it is impossible to predict what will be executing at the time the power fail occurs, the **atpfail** function must be coded to have no semaphore ramifications.

RETURN VALUE

As released, **atpfail** returns 1 under all conditions.

If you define your own **atpfail** function, it will have the return value you define. A common use of **atpfail** is to "ride out" the power failure; if it is still running after 5 seconds, it indicates a backup power supply has taken over and the system is still up. If **atpfail** returns any value, the system will issue the following console error message: "`AC - FAIL`".

LEVEL

Base or Interrupt

SOURCE FILE

stub/atpfail.c (code should be put in */usr/src/uts/realix/custom/atpfail.c*)

SEE ALSO

atpanic(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 both be within kernel address space. No range checking is done. If an address outside kernel address space is selected, the system will panic.

Note that **bcopy** should never be used to move data in or out of a user buffer because it has no provision for handling page faults (use **copyin**(D3X) and **copyout**(D3X) instead). 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 a user buffer when it is swapped out, the system will crash. Because kernel space is never swapped out, it is safe to use **bcopy** to move data within kernel space.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

Under all conditions, 0 (zero) is returned.

LEVEL

Base or Interrupt

SOURCE FILE *ml/*/misc.s*

SEE ALSO *KPG*, "Synchronized I/O Operations"
copyin(D3X), **copyout**(D3X), **fubyte**(D3X), **fuword**(D3X), **iomove**(D3X), **subyte**(D3X), **suword**(D3X)

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).
- ❑ Otherwise, the I/O operation is a write request; the data is copied from a buffer to the RAM disk (line 10).

The **bcopy** function is used because 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) {
 7           bcopy(&ramdblks[bp->b_blkno][0], bp->b_un.b_addr, bp->b_bcount);
 8   }

 9   else {
10           bcopy(bp->b_un.b_addr, &ramdblks[bp->b_blkno][0], bp->b_bcount);
11   }
```

NAME

bmemalloc – allocate memory

SYNOPSIS

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

char *
bmemalloc(siz)
int siz;
```

ARGUMENTS

siz	the number of bytes to be allocated

DESCRIPTION

This function allocates a specified number of bytes of memory. The normal return value is the kernel virtual address of the allocated space. Allocated space is virtually, but not physically, contiguous.

Using **bmemalloc** does not guarantee any alignment of allocated space.

SEMAPHORE RAMIFICATIONS

No spin locks can be held when calling **bmemalloc**.

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)

SOURCE FILE

sys/sysmacros.h

SEE ALSO

KPG, "Memory Management"
bmemfree(D3X)

NAME

bmemfree – free allocated memory

SYNOPSIS

```
bmemfree(vaddr, siz)
char * vaddr;
int siz;
```

ARGUMENTS

vaddr base virtual address of memory to be released, which is returned from **bmemalloc**

siz number of bytes to be released; must be the same as the *siz* argument used with the associated call to **bmemalloc**

DESCRIPTION

This function releases memory or performs garbage cleanup to free allocated memory for reuse. This function is called after **bmemalloc**(D3X) to free allocated memory.

SEMAPHORE RAMIFICATIONS

No spin locks can be held when calling **bmemfree**.

RETURN VALUE

None.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

sys/sysmacros.h

SEE ALSO

KPG, "Memory Management"
bmemalloc(D3X)

NAME

bprobe, sprobe, lprobe – access an address with recovery from errors

SYNOPSIS

```
int
bprobe(addr, val)
char * addr;
int val;
```

The synopses of **sprobe** and **lprobe** are the same as the synopsis of **bprobe**.

ARGUMENTS

addr base virtual address to be tested

val specifies a read probe or write probe. If *val* is negative, **bprobe** reads the specified address; otherwise, **bprobe** writes *val* to *addr*.

DESCRIPTION

This function typically is used during driver initialization to determine if the board associated with the driver is installed at a given address. If the value of the second argument (*val*) is less than 0, **bprobe** reads the byte at the address given in the first argument (*addr*); otherwise, **bprobe** writes the non-negative value of *val* to that address. In either case, a bus error occurs if the addressed location is not configured in the system. The bus handler recognizes that the bus error is a result of a **bprobe** and ensures that **bprobe** returns the appropriate value.

sprobe and **lprobe** are functionally the same as **bprobe**. The three variations are provided to accommodate devices that respond only to an access of the appropriate size. Whereas **bprobe** operates on a byte (8 bits of data), **sprobe** accesses 16 bits (a short value), and **lprobe** accesses 32 bits (a long value).

SEMAPHORE RAMIFICATIONS

No spin locks can be held when calling **bprobe**, **sprobe**, or **lprobe**.

CAVEATS

It is strongly recommended that **bprobe**, **sprobe**, or **lprobe** be called only as part of driver initialization, before any driver processes are running. Once processes are running, these functions should not be called because, if the address being probed is a non-existent location, realtime performance can be impacted. Attempting to access a non-existent location will lock up the processor and VMEbus until the bus times out (producing a bus error) and the call fails.

RETURN VALUE

If the device is present (a bus error does not occur), **bprobe**, **sprobe**, or **lprobe** returns 0. If the device is not present (a bus error occurs), **bprobe**, **sprobe**, or **lprobe** returns 1.

LEVEL	Base Only (Do not call from an interrupt routine)
SOURCE FILE	*ml/*/misc.s*

NAME

brelse – return buffer to the bfreelist

SYNOPSIS

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

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

ARGUMENTS

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 is called after the driver function is finished with the buffer. It returns a buffer to the `bfreelist` pool of free buffers as a function of B_AGE, unblocks any processes that may be waiting for a free buffer, then unlocks a semaphore to allow other processes to lock the buffer.

If B_AGE is set, the buffer will be reused before other buffers in the system. B_AGE should be set when you know that the data in the buffer will not be needed by other processes.

CAUTION

The flags in the **b_flags** *member of the* buf*(D4X) structure must have appropriate settings when* **brelse** *is called. Otherwise, the disk may be corrupted and the system may panic.*

If the buffer was allocated with **geteblk***(D3X) or* **getnblk***(D3X), the buffer is not assigned to any particular device and block number. After* **brelse** *executes, the buffer will be reassigned to some other use. However, if the B_DELWRI flag is set, the system will attempt to write the data in the buffer to the device and block number specified in the appropriate* buf *fields.*

b_flags *should be treated as shown in the example that follows.*

SEMAPHORE RAMIFICATIONS

No spin locks can be locked when invoking **brelse**. Any necessary locks are handled by **geteblk**(D3X) or **getnblk**(D3X), which should have been called before **brelse**.

RETURN VALUE

brelse does not return a 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

SOURCE FILE

os/bio.c

SEE ALSO

KPG, "Synchronized I/O Operations"
geteblk(D3X), **getnblk**(D3X), **clrbuf**(D3X), **iowait**(D3X), **preiowait**(D3X), `buf`(D4X)

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, relevant fields in the buffer header are set up, and the physical I/O is scheduled by calling the driver's **strategy** routine (line 7).
- ❑ The driver waits for the completion of the physical I/O operation (line 8).
- ❑ **b_flags** is set to ensure that the system does not subsequently attempt to write the data in the buffer to disk (line 9). Clearing all the flags except B_BUSY is not required on the REAL/IX Operating System because B_DELWRI should not have been set by any code in this example. However, for portability considerations it is good practice to include this line in your code.
- ❑ **b_flags** is set to ensure the buffer is reused again quickly (line 10). This optimization ensures that possibly useful buffers in the cache are not reused before this buffer, which is no longer needed.
- ❑ The unblocked base level portion of the driver then releases the buffer (line 11).
- ❑ When the I/O operation is finished, the driver's interrupt routine calls **iodone**(D3X) to unblock (line 15).
- ❑ Note that any error setting within the buffer will have caused **iowait** (line 8) to place the error code in the `u_area`. It is not necessary for the driver to check buffer fields explicitly

```
register struct buf *bp;

  ⋮

bp = geteblk;
clrbuf(bp);

/* Copy data to allocated buffer and    */
/* schedule physical I/O request with device */

xxstrategy(bp);
iowait(bp);
bp->b_flags &= B_BUSY;
bp->b_flags |= B_AGE | B_STALE;
brelse(bp);

  ⋮

xxintr(subvec); {

              ⋮

        iodone(bp);
}
```

btoc(D3X)

NAME

btoc, btoct – convert bytes to clicks (memory pages)

SYNOPSIS

```
unsigned
btoc(bytes)
unsigned bytes;
```

The synopsis of **btoct** is the same as the synopsis of **btoc**.

ARGUMENTS

bytes number of bytes

DESCRIPTION

These macros return the number of memory pages (clicks) that are needed to contain a specified number of bytes. **btoc** rounds up to the next page and can be used to determine the number of pages required to hold the specified number of bytes; **btoct** (truncated) rounds down and is used to determine the page on which the number of bytes ends. For example, if the page size on your system is 4096 bytes,[1] then **btoc(14384)** returns 4 and **btoct(14384)** returns 3. **btoc(0)** and **btoct**(0) both return 0.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

A non-negative value is always returned.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sysmacros.h

SEE ALSO

ctob(D3X)

[1] The page size used by the REAL/IX Operating System varies depending on the hardware platform on which it runs. Refer to the Release Notes shipped with your system.

NAME

bzero – clear memory for a specified 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 0s (zeroes).

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

bzero returns 0 whether or not it is successful.

LEVEL

Base and Interrupt

SOURCE FILE

ml//misc.s*

SEE ALSO

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

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 handles 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 more information about 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**(D3X) 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_can** queue. **canon** is called from the **ttread** line discipline routine to do this.

SEMAPHORE RAMIFICATIONS

Drivers that use **canon** must be installed under a compatibility mode.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/tty.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
ttread(D3X), **ttin**(D3X)

RETURN VALUE

In general, **canon** blocks 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. Specifically, **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).

EXAMPLE This excerpt from **ttread**(D3X) uses **canon** from a driver **read** routine.

```
ttread(tp)
register struct tty *tp;
{
    register struct clist *tq;

    tq = &tp->t_canq;

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

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

     {
     /* transfer characters to user data space from canq */
     }
}
```

NAME

cintrctl – connected interrupt I/O control operations (IOCTLs)

SYNOPSIS

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

int cintrctl(cid, command, arg)
int cid, command;
struct cintrio *arg;
```

ARGUMENTS

cid identifies the connected interrupt structure on which to perform the *command*. *cid* is returned by a previous call from the **cintrget**(D3X) function.

command the connected interrupt control function to be performed, passed from user-level process's **ioctl**(2) call.

arg pointer to a `cintrio(4)` data structure that contains additional information needed by this *command*, passed from user-level process's **ioctl**(2) call (optional; not all commands require an *arg*).

DESCRIPTION

This function is used in the driver's **ioctl**(D2X) routine to implement all connected interrupt IOCTL commands listed on the `cintrio(4)` manual page except CI_CONNECT (which is implemented with the **cintrget**(D3X) function). The functions implemented are:

CI_UCONNECT
disconnect the process associated with the connected interrupt identifier (*cid*). The *cid* is removed and the associated data structure is released. This function is equivalent to **cintrelse**(D3X).

CI_SETMODE
switch the bit of the **ci_flags** member of the structure. If set to CINTR_PERIODIC, the user-level process is notified of all device interrupts; if not set, the user-level process is notified of one interrupt at a time; subsequent interrupts are ignored until the previous one is acknowledged with the CI_ACK command.

CI_ACK acknowledge the last delivered device interrupt (ignored if the CINTR_PERIODIC flag is set).

CI_STAT populate *arg* with the values currently assigned to *cid*. *arg* must point to a user address.

For more information about using these IOCTL commands in user-level programs, refer to **cintrio**(7) and to the *Programmer's Guide*.

SEMAPHORE RAMIFICATIONS

No spin locks can be locked when invoking **cintrctl** with the CI_UCONNECT function.

RETURN VALUE

On success, a value of 0 is returned. Otherwise, a value of −1 is returned and **u.u_error** is set to indicate the error. **cintrctl** will set **u.u_error** to EINVAL, EFAULT, or ENODEV.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/cintr.c

SEE ALSO

KPG, "Interrupts"
cintrget(D3X), **cintrnotify**(D3X), **cintrelse**(D3X)
evctl(2), **evget**(2), **evrcv**(2), **evrcvl**(2), **evrel**(2), **cintrio**(4), **cintrio**(7)

NAME

cintrelse – release a connected interrupt identifier

SYNOPSIS

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

int cintrelse(cid)
int cid;
```

ARGUMENTS

cid identifies the connected interrupt structure to be released. *cid* is returned by a previous call from the **cintrget**(D3X) function.

DESCRIPTION

This function is used in the driver's **close**(D2X) routine to disconnect the process associated with the connected interrupt identifier *cid* (if it was not previously disconnected with a CI_UCONNECT **cintrctl**(D3X) command), remove the connected interrupt identifier, and release the data structure associated with it.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **cintrelse**.

RETURN VALUE

If successful, 0 is returned. Otherwise, a value of −1 is returned and **u.u_error** is set to EINVAL.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/cintr.c

SEE ALSO

KPG, "Interrupts"
cintrctl(D3X), **cintrnotify**(D3X)

NAME

cintrget – connect the driver to a `cintrio`(4) structure

SYNOPSIS

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

int cintrget(key, arg, flg)
int key, flg;
struct cintrio *arg
```

ARGUMENTS

key the connected interrupt key. By convention, this is the device number (major and minor number concatenated), although any value can be used.

arg pointer to a `cintrio`(4) data structure that contains additional information needed by this *command*, passed from user-level process's **ioctl**(2) call.

flag CINTR_EXCL if exclusive access is required for this key; otherwise, 0.

DESCRIPTION

This function is called in the driver's **ioctl**(D2X) routine to implement the connected interrupt CI_CONNECT IOCTL command. It returns the connected interrupt identifier associated with *key*. On each successful call, **cintrget** creates a connected interrupt identifier and an associated `cintr`(D4X) data structure, and populates the `cintr` structure with information from the associated user-level `cintrio`(4) structure.

SEMAPHORE RAMIFICATIONS

No spin locks can be locked when invoking **cintrget**.

RETURN VALUE

Upon success, a non-negative integer (the connected interrupt identifier) is returned. Otherwise, a value of –1 is returned and **u.u_error** is set to EPERM, EINVAL, EFAULT, or ENOSPC to indicate the error.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/cintr.c

SEE ALSO

KPG, "Interrupts"
cintrctl(D3X), **cintrnotify**(D3X), **cintrelse**(D3X)
evctl(2), **evget**(2), **evrcv**(2), **evrcvl**(2), **evrel**(2), **cintrio**(4), **cintrio**(7)

NAME

cintrnotify, CINTRNOTIFY – notify the user-level process of an interrupt

SYNOPSIS

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

void cintrnotify(cid, dataitem)
int cid, dataitem
```

The synopsis of **CINTRNOTIFY** is the same as the synopsis of **cintrnotify**.

ARGUMENTS

cid identifies the process to be notified of the interrupt. *cid* is returned by a previous call from the **cintrget**(D3X) function.

dataitem if the notification method for this *cid* is CINTR_EVENTS, this is the *dataitem* to be written to the evt structure associated with this connected interrupt; otherwise is unused.

DESCRIPTION

This function is used in the driver's **intr**(D2X) routine to notify the user-level process associated with the connected interrupt identifier *cid* of an interrupt. The notification method used is that which was requested by the CI_CONNECT command for identifier *cid*.

CINTRNOTIFY is an inline (macro) version defined in *sys/cintrio.h*. It provides the same functionality as **cintrnotify** and takes the same arguments, but is faster.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **cintrnotify**.

RETURN VALUE

cintrnotify returns a status code as follows:

0	no errors
EINVAL	no process is connected to the interrupt
EBUSY	interrupt was set up as a one-shot and has not yet been acknowledged
ENOSPC, EAGAIN	error code from **send_event**(D3X)

CINTRNOTIFY does not return a value under any conditions.

LEVEL

Interrupt Only (Do not call from a base level routine)

SOURCE FILE

os/cintr.c

SEE ALSO

KPG, "Interrupts"
cintrctl(D3X), **cintrget**(D3X), **cintrelse**(D3X)
evctl(2), **evget**(2), **evrcv**(2), **evrcvl**(2), **evrel**(2), **cintrio**(4), **cintrio**(7)

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;
```

ARGUMENTS

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).

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base and Interrupt

SOURCE FILE

os/bio.c

SEE ALSO

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

EXAMPLE

See the example for **geteblk**(D3X) for an example of **clrbuf**.

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, arg;
```

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 can be used to continue other messages or display informative messages not connected with an error during system initialization. It is not recommended outside **init**(D2X) routines because other code could interrupt this code between the first and second lines of the error. Moreover, using CE_CONT makes it more difficult to **grep** for all WARNING and NOTICE messages in the */usr/adm/putbuf* file.

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 system cannot continue to function. If the error is recoverable, or not essential to continued system operation, CE_PANIC should not be specified.

CAUTION

An invalid value for level *will panic the system when* **cmn_err** *executes.*

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**(3S) strings. The special characters are as follows:

! directs the output of the string only to the **putbuf**, a circular array in memory used to store messages. The messages usually are read by **putbuf**(1) using */dev/osm* and are written to a log file, usually */usr/adm/putbuf*.

^ displays the message only on the console

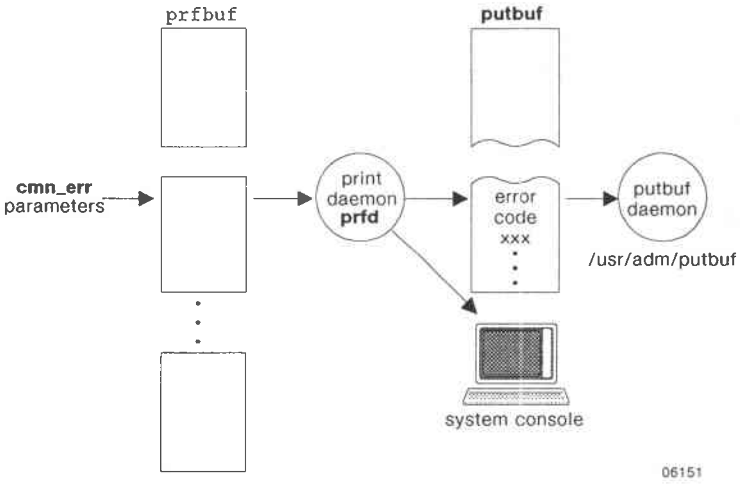
If a special character is omitted from the first string position, the message is directed to both the **putbuf** and the console. Except for CE_CONT, **cmn_err** appends "\n" to each *format* whether displaying information about the console and/or writing the format message to **putbuf**. CE_CONT messages are printed as written (no "\n" is appended).

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 the **putbuf**. Up to 10 arguments can be printed.

Note that `%s` is *not* a valid conversion specification for local stack variables. Note also that **cmn_err** does not accept length specifications in conversion specifications. For example, `%3d` is ignored.

DESCRIPTION

The **cmn_err** function is used to write error and informational messages to the console and/or the **putbuf** structure. On the REAL/IX Operating System, **cmn_err** messages are written to the prfbuf structure,[1] and the print daemon (**prfd**)[2] moves messages from prbuf to the console, the **putbuf**, or both (see figure).



Use **cmn_err** to notify the administrator of specific actions required (such as mounting a tape on the driver or adding paper to the printer) or to provide information about device conditions that may eventually cause serious system problems (for instance, if retries are required to complete the operation, the device may need repair, even though the operation eventually succeeded). **cmn_err** can also be used for messages that allow you to trace the progress through the driver code during the debugging stage or that report perform-

[1] By default, prfbuf has 100 entries and the **putbuf** is 2000 bytes long. If **cmn_err** messages are being lost because prfbuf is too small, the message 'cmn_err: too many messages, *xx* lost' is displayed. Messages may also be lost if the size of the **putbuf** is too small; however, no message is displayed in this case. You can increase the size of prfbuf by modifying the MAXPRBUFS kernel parameter in **sysgen**(1M); you can increase the size of the **putbuf** by modifying the PUTBUFSZ kernel parameter in **sysgen**. If you increase the value of either one, you should increase the value of the other one, too.

[2] **prfd** does not execute during kernel initialization or when the system panics. In these cases, **cmn_err** messages are written directly to the console and the **putbuf**. With superuser privileges, you can force **cmn_err** messages to be written directly to the console and the **putbuf** (by means of the RLXPRFCTL command of **sysrealix**(2)). This method guarantees that no messages are lost, but may have an adverse impact on real time performance (interrupt latency). This method may be useful during driver development, but is *not* recommended when running a production system.

ance statistics (such as the amount of time required to complete the I/O operation) when doing performance testing.

If CE_PANIC is set, **cmn_err** stops the machine. This is used often for debugging (because panicking the machine enables you to save a copy of memory that can be analyzed), but should be used very carefully in production drivers. Drivers should avoid panicking the system except when it is clear that the kernel is corrupted or some other condition exists that makes it dangerous for the system to continue to run.

SEMAPHORE RAMIFICATIONS

None.

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, the system will attempt to print both messages with an indication of the order in which the panic calls occurred.

LEVEL

Base or Interrupt

SOURCE FILE

os/prf.c

SEE ALSO

KPG, "Process Notification"
print(D2X), **atpanic**(D3X)

EXAMPLES

The first code example below illustrates how **cmn_err** is used to provide information that a routine has been called during the testing phase. Note that, because the "%x" conversion character is used, the minor/major number of the device will be printed in hexadecimal.

```
      register struct device *rp;
      rp = xx_addr[(minor(dev) >> 4) & 0xf)];
   #if TEST
      cmn_err(CE_NOTE, "xx_open routine called - dev = 0x%x",
      dev);
   #endif
```

The next code fragment shows that the **cmn_err** function can:

- □ record tracing and debugging information in the **putbuf** (lines 12 – 13)
- □ display information about the device on the system console (line 15)
- □ stop the system if a required device malfunctions (line 19)

```
 1   struct device {                    /* Physical device registers layout */
 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   };
 8   extern struct device xx_addr[];        /* Physical device registers */
 9   extern int        xx_cnt;              /* Number of physical devices */

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

12   cmn_err(CE_NOTE, "!xx_open function called - dev = 0x%x", dev);
13   cmn_err(CE_CONT, "! flag = 0x%x", flag);
14   if ((rp->status & POWER)  == OFF) {
15           cmn_err(CE_WARN, "^xx_open: Power is OFF on device %d port %d",
16   }
17   ((dev>>4) & 0xf), (dev &0xf));
18   if (rp->error == BADVTOC && dev == rootdev){
19           cmn_err(CE_PANIC, "xx_open: Bad VTOC on root device");
20   }
```

NAME

comp_aio – indicates that an asynchronous I/O operation has completed

SYNOPSIS

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

comp_aio(areq, byte_cnt, status)
areq_t *areq;
int byte_cnt, status;
```

ARGUMENTS

areq pointer to the areq(D4X) structure being used for this operation

byte_cnt number of bytes transferred; must be −1 if status is not 0

status indicates whether the operation completed successfully (0) or unsuccessfully (non-zero)

DESCRIPTION

comp_aio updates the areq(D4X) structure to indicate that an asynchronous I/O operation has completed. If an aiocb(4) structure was given in the initiating **aread**(2) or **awrite**(2) call, **comp_aio** populates the **rt_errno** and **nobytes** members of the aiocb. If required, the *eid* in the areq structure is posted to the associated user-level process.

SEMAPHORE RAMIFICATIONS

No spin locks should be set when calling **comp_aio**. In particular, **areq->p->p_lock** must be unlocked.

RETURN VALUE

comp_aio does not return a value under any conditions. The *status* argument should hold an appropriate error code for unsuccessful operations (refer to **aread**(2) and **awrite**(2) for a list of error codes that are anticipated by the system calls).

LEVEL

Base or Interrupt

SOURCE FILE

os/aio.c

SEE ALSO

KPG, "Miscellaneous I/O Operations"
aio(D2X), **comp_cancel_aio**(D3X), areq(D4X)
aread(2), **awrite**(2), aiocb(4)

NAME

comp_cancel_aio – indicate that an asynchronous I/O operation has been canceled

SYNOPSIS

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

comp_cancel_aio(areq)
areq_t *areq;
```

ARGUMENTS

areq pointer to the areq(D4X) structure being used for this operation

DESCRIPTION

When the **aio**(D2X) routine is called with the ACANCEL *cmd*, it is up to the driver whether the asynchronous operation is really to be canceled. If so, the driver calls **comp_cancel_aio** and returns ACANYES to the **aio** routine.

comp_cancel_aio updates the areq(D4X) structure to indicate that an asynchronous I/O operation is no longer in progress. If there was an aiocb(4) structure given in the initiating **aread**(2) or **awrite**(2) call, then the **rt_errno** member of the aiocb(4) is set to ECANCELLED and the **nobytes** member is set to −1.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **comp_cancel_aio**. In particular, `areq->p->p_lock` must be unlocked.

RETURN VALUE

comp_cancel_aio does not return a value under any conditions.

LEVEL

Base or Interrupt (Usually called from base level)

SOURCE FILE

os/aio.c

SEE ALSO

KPG, "Miscellaneous I/O Operations"
aio(D2X), **comp_aio**(D3X), areq(D4X)
acancel(2), **aread**(2), **awrite**(2), **aiocb**(4)

NAME

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

SYNOPSIS

```
int
copyin(userbuf, driverbuf, count)
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)

count 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 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.

SEMAPHORE RAMIFICATIONS

No locks should be held when calling **copyin**.

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)

SOURCE FILE

ml//userio.s*

SEE ALSO

KPG, "Synchronized I/O Operations"
bcopy(D3X), **copyout**(D3X), **fubyte**(D3X), **fuword**(D3X), **iomove**(D3X), **subyte**(D3X), **suword**(D3X)

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 and an error code is returned (lines 6–8). Otherwise, the pointer to the user data area is advanced to the next starting byte of data to be copied (line 11), and the remaining byte count is updated (line 12).

```
 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);

 4          if (copyin(u.u_base, addr, cnt) == -1)
 5          {
 6                mfree(xx_map, cnt, (uint)addr);
 7                u.u_error = EFAULT;
 8                return;
 9          }

10            ⋮

11          u.u_base += cnt;
12          u.u_count -= cnt;
13   }
```

NAME

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

SYNOPSIS

```
copyout(driverbuf, userbuf, count)
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)
count	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**(D2X) or **write**(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, 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 decrease 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.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **copyout**.

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 table, 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)

SOURCE FILE

ml//userio.s*

SEE ALSO

KPG, "Synchronized I/O Operations"
bcopy(D3X), **copyin**(D3X), **fubyte**(D3X), **fuword**(D3X), **iomove**(D3X), **subyte**(D3X), **suword**(D3X)

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 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:
18           if(copyout(rp,(struct device *)arg,sizeof(struct device)) == -1) {
19               u.u_error = EFAULT;
21               break;
20           }
22         ⋮
23      }
24     ⋮
```

NAME

cpass – get next character from user's write call

SYNOPSIS

`cpass()`

ARGUMENTS

None.

DESCRIPTION

cpass picks up the next character from location **u.u_base** in the current user(D4X) structure, and updates the **u.u_base**, **u.u_count**, and **u.u_offset** members.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

If successful, **cpass** returns the next character. If **u.u_count** is 0 (meaning there are no characters to be written), **cpass** returns −1. If there is an access fault (**u.u_base** points outside the user's address space), **cpass** returns −1 and sets **u.u_error** to EFAULT.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/move.c

SEE ALSO

passc(D3X), user(D4X)

EXAMPLE

This example comes from the kernel code that allows users to write to the internal **putbuf** structure via */dev/osm*.

```
extern sema_t putbuf_sema;      /* blocking semaphore for putbuf structure */
extern putbufsz;                /* size of last offset read from */
extern putbufndx;               /* next position to write to */

osmwrite()
{
      register int cc;

      while ((cc = cpass()) >=0)
            putbuf[putbufndx++ % putbufsz] = cc;
}
```

NAME

cpsema, rcpsema, pcpsema – lock semaphore for a resource if the resource is available

SYNOPSIS

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

val = cpsema(sem_addr, flags)
sema_t *sem_addr;
int flags;
```

The synopses of **rcpsema** and **pcpsema** are the same as the synopsis of **cpsema**.

ARGUMENTS

sem_addr semaphore to lock

flags flags; valid values are:

0	Boosting algorithm should not be used.
SEMRTBOOST	Apply a boosting algorithm that temporarily boosts the priority of lower priority process when it holds the semaphore if the semaphore is needed by a higher priority realtime process. This flag should only be applied to semaphores that are expected to be used by realtime processes after their initialization time processing.

DESCRIPTION

The **cpsema** family of macros locks the semaphore for a resource by decrementing its value, similar to the **psema** family of macros. The difference between the two is that **cpsema** locks a resource only if it is immediately available; if **cpsema** finds that the semaphore is already locked (a value of 0 or less), it returns without changing the value of the semaphore.

Note that, if the SEMRTBOOST flag is used, all calls for that semaphore (**psema**, **cpsema**, and **vsema**) must also use the SEMRTBOOST flag. This restriction is necessary to ensure that the boosting algorithm is reliable.

Semaphores locked with a member of the **cpsema** family can be unlocked with any member of the **vsema** family of macros.

The **rcpsema** and **pcpsema** macros are available for optimizing driver performance. **rcpsema** can be used if interrupts are already disabled with **spsema**(D3X); **pcpsema** can be used if interrupts are fully enabled.

SEMAPHORE RAMIFICATIONS

Drivers that call **cpsema** must be installed fully semaphored. A spin lock may be held when calling **cpsema**.

RETURN VALUE

If the value of the semaphore is greater than zero (unlocked) on entry, **cpsema** returns 1, indicating that it got the resource. Otherwise, **cpsema** returns 0.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sema.h

SEE ALSO

KPG, "Synchronization"
cvsema(D3X), **decsema**(D3X), **incsema**(D3X), **initsema**(D3X), **psema**(D3X), **psvsema**(D3X), **valulock**(D3X), **valusema**(D3X), **vsema**(D3X)

NAME

ctob – convert clicks to bytes

SYNOPSIS

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

unsigned
ctob (clicks)
unsigned clicks;
```

ARGUMENTS

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 on your system is 4096 bytes, **ctob(2)** returns 8192.[1] **ctob(0)** returns 0.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

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

LEVEL

Base or Interrupt

SOURCE FILE

sys/sysmacros.h

SEE ALSO

btoc(D3X)

[1] The page size used by the REAL/IX Operating System varies depending on the hardware platform on which it runs. Refer to the Release Notes shipped with your system.

cvsema(D3X)

NAME

cvsema, rcvsema, pcvsema – unlock semaphore for a resource if a process is waiting or make resource available

SYNOPSIS

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

cvsema(sem_addr)
sema_t *sem_addr;
```

The synopses of **rcvsema** and **pcvsema** are the same as the synopsis of **cvsema**.

ARGUMENTS

sem_addr identifies the semaphore to be unlocked; must correspond to the *sem_addr* used to lock the resource.

DESCRIPTION

The **cvsema** family of macros increments a semaphore value (thus unblocking a process) only if a process is waiting for the semaphore (in other words, the semaphore value is less than 0). If the semaphore value is greater than or equal to 0, the **cvsema** macros do nothing.

cvsema is used with semaphores that are initialized to 0 to unblock any processes that are suspended. **cvsema** cannot be used if the **psema** call that blocked the process used any flags. The **cvsema** macros are not commonly used in drivers. An example of their use is the clock interrupt, which does a **cvsema** to unblock a process that may have done a **psema**. Also system daemons that have been blocked with a **psema** call are unblocked with **cvsema**.

The **rcvsema** and **pcvsema** macros are faster versions of **cvsema**. **rcvsema** can be used if all interrupts are guaranteed to be disabled; **pcvsema** can be used if all interrupts are guaranteed to be enabled.

NOTE

This is not a reliable mechanism because the process to be unblocked may not yet have issued a ***psema*** *(for example, it may not have run due to other, high-priority processes being scheduled). However, this is a convenient way to periodically unblock processes.*

SEMAPHORE RAMIFICATIONS

Drivers that call **cvsema** must be installed fully semaphored.

RETURN VALUE

The **cvsema** macros do not return a value under any conditions.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sema.h

SEE ALSO

KPG, "Synchronization"
cpsema(D3X), **decsema**(D3X), **incsema**(D3X), **initsema**(D3X), **psema**(D3X), **psvsema**(D3X), **spsema**(D3X), **svsema**(D3X), **valulock**(D3X), **valusema**(D3X), **vsema**(D3X)

NAME

dcachclr – flush the virtual čache, if present

SYNOPSIS

```
dcachclr()
```

ARGUMENTS

None.

DESCRIPTION

dcachclr flushes the virtual data cache on the CPU, if present. The function performs no action if there is no virtual cache. Flushing the cache ensures that stale data is eliminated from the data cache. This may be required because:

- ❑ The cache can contain data that has been mapped via a virtual address, so if different pieces of data are referenced by two different processes, each using the same virtual addresses, it can get out of synchronization.
- ❑ A controller board may have written directly into main memory, and the data cache must be flushed to be synchronized with main memory. For controllers that read and write global memory, there are times when it is crucial that the data cache is synchronized with main memory.

An **intr**(D2X) routine or other interrupt handler can be **sysgen**ed to automatically flush the onboard data cache after it executes, but if the interrupt handler needs to look at data in the cache that could be stale, it needs to explicitly flush the cache. The **dcachclr** function is necessary for processors that have a virtual cache to ensure that cache contents are not stale.

Drivers that use **dcachclr** must be compiled with a **sed**(1) script. The *custom/custom.mk* file handles this automatically.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

scsi/scsicmd.h

SEE ALSO

intr(D2X)

NAME 386 debug – invoke the kernel debugger

SYNOPSIS

```
debug()
```

DESCRIPTION

debug invokes the kernel debugger by a trap 3. This function allows kernel programmers to debug and troubleshoot by calling the kernel debugger from their code.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE None.

LEVEL Base or Interrupt

SOURCE FILE *sys/inline.h*

SEE ALSO *KPG*, **kdb**(1M)

NAME

decsema, rdecsema, pdecsema – decrement a semaphore value for a resource by 1

SYNOPSIS

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

decsema(sem_addr)
sema_t *sem_addr;
```

The synopses of **rdecsema** and **pdecsema** are the same as that of **decsema**.

ARGUMENTS

sem_addr identifies the semaphore to be decremented

DESCRIPTION

The **decsema** family of macros decrement by one the value of the semaphore specified by *sem_addr*. The are used to manipulate counters (such as the number of I/O operations in progress) for statistics, and should not be used for synchronization or exclusion.

rdecsema and **pdecsema** provide functionality similar to that of **decsema**, but are faster. **rdecsema** can be used when all interrupts are disabled with a spin lock; **pdecsema** can be used when all interrupts are guaranteed to be enabled.

SEMAPHORE RAMIFICATIONS

Drivers that call **decsema** should be installed fully semaphored.

RETURN VALUE

The **decsema** macros do not return a value under any conditions.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sema.h

SEE ALSO

incsema(D3X)

NAME

DELAY – delay by spinning when no clock timing is available

SYNOPSIS

```
DELAY(microseconds)
```

ARGUMENTS

microseconds the amount of time to suspend the code. This is converted internally into the proper spin count.

DESCRIPTION

DELAY provides a way of delaying a process for a specified amount of time, independent of clock interrupts. This provides finer resolution than **delayfs**(D3X) and **delay**(D3X).

Defined constants can be used with DELAY to convert other time measures

to microseconds:

MS_TO_US	milliseconds to microseconds
HS_TO_US	1/100 seconds to microseconds
TS_TO_MS	1/10 seconds to microseconds
SECONDS_TO_US	seconds to microseconds

A millisecond is 1/1000 second; a microsecond is 1/1,000,000 second; a nanosecond is 1/1,000,000,000.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sysmacros.h (defines **DELAY** macro*); sys/param.h* (defines constants)

VMEbus

io/vme/mvmecpu.c (defines clock rate assumptions for supported processors)

NAME

delay, delayfs - delay process execution for a specified number of clock cycles

SYNOPSIS

```
delay(ticks)              /* compatibility mode drivers */
int ticks

delayfs(ticks)            /* fully semaphored drivers */
int ticks
```

ARGUMENTS

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

DESCRIPTION

Occasionally, you may need to wait a given period of time until work is available. The **delay** and **delayfs** functions provide the wait time. The exact time interval that the delay takes effect cannot be guaranteed, but the value given is a close approximation.

SEMAPHORE RAMIFICATIONS

delay is used only with drivers installed for semaphoring on the driver entry (compatibility modes); drivers that are fully semaphored should use the **delayfs**(D3X) function instead.

No spin locks should be held when calling **delayfs**. **delay** can be used only in drivers installed under the compatibility modes.

RETURN VALUE

None.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/clock.c

SEE ALSO

KPG, "Synchronization"
iodone(D3X), **iowait**(D3X), **sleep**(D3X), **timeout**(D3X), **ttywait**(D3X), **untimeout**(D3X), **wakeup**(D3X)

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 for a specific period of time (line 14) 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 */
12   {                      /* display message & ring bell on system console */
13      cmn_err(CE_WARN, "^xx_write: NO PAPER in printer %d 07", (dev & 0xf));
14      delay(60 * HZ);                 /* Wait one minute and try again */
15   }                                  /* endwhile */
```

disable(D3X)

NAME

disable – disable interrupts for the processor on which code is executing

SYNOPSIS

`disable()`

ARGUMENTS

None.

DESCRIPTION

disable disables all interrupts for the processor on which code is executing. **spl***(D3X) and **spsema**(D3X) call **disable** internally, and usually it is better to use these functions than to call **disable** directly. **disable** is useful for protecting a local resource (such as a board) with less overhead than the other functions entail.

CAUTION

***disable** does not protect global data structures in a multiprocessor environment. Only spin locks can guarantee that data structures will be protected. Do not use **disable** in drivers that are written for or that may eventually be ported to a multiprocessor configuration.*

386

CAUTION

*On 386/486-based systems, we strongly recommend using the **popsr**(D3X) and **pushsrdisable**(D3X) kernel functions instead the of **enable**(D3X) and **disable** kernel functions.*

Disabling interrupts for long periods of time will degrade general system performance.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os//interrupt.c*

386

sys/inline.h

SEE ALSO

enable(D3X), **spl**(D3X), **spsema**(D3X), **svsema**(D3X), **popsr**(D3X), **pushsrdisable**(D3X)

EXAMPLE

The sample driver *avme9510.c* uses **disable/enable** in its **close**(D2X) routine to protect the code that disables the timer and interrupts from the board. If an interrupt were received in the middle of this code, it would generate a spurious interrupt that might corrupt the kernel. To modify this code for a multiprocessor, **disable** would be changed to **spsema** and **enable** would be changed to **svsema**.

See the *sys/avme9510.h* header file for a definition of the structure and corresponding register fields that are used.

```
a950close(dev)
     dp = (struct a9510_dev *) a950_adr[ctrl];

       ⋮

     disable();
     dp->a_control &= ~BC_CNTEN;
     dp->a_status &= ~A_ENABLE;
     enable();
```

NAME

disjointio -- get physical location of user virtual memory

SYNOPSIS

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

int disjointio(bp, djntptr, szdjnt, maxtc);
struct buf *bp;
struct djntio *djntptr;
unsigned szdjnt maxtc;
```

ARGUMENTS

bp	pointer to buffer header
djntptr	disjoint array for discontiguous pages
szdjnt	size of disjoint array
maxtc	maximum transfer count in bytes for each TA/TC pair; must be multiple of the page size[1]

The following members of buf(D4X) are implicit arguments to **disjointio**:

b_un.b_addr	virtual address of buffer in user space
b_bcount	buffer size, in bytes
b_flags	sets B_READ and, if appropriate, B_AIO

DESCRIPTION

disjointio finds the physical location of an area of user virtual memory. The physical memory may not be contiguous; it is described by a sequence of physical address 1-byte count pairs called TA/TC pairs (for transfer address, transfer count, on the assumption that the mapping is for the purposes of an I/O transfer).

The virtual memory is described by the **b_un.b_addr** and **b_bcount** members of the buf(D4X) structure pointed to by *bp*. *djntptr* points to an area where the TA/TC pairs are to be recorded, and *szdjnt* gives the maximum number of TA/TC pairs that can fit in this area.

disjointio does not necessarily generate a TA/TC pair for every page of physical memory; if the pages are contiguous, they can be described by a single TA/TC pair. The *maxtc* parameter controls how large a transfer count is allowed in one TA/TC pair. This degree of control is provided because certain devices have a fixed limit for the byte count in a TA/TC pair.

[1] The page size used by the REAL/IX Operating System varies depending on the hardware platform on which it runs. Refer to the Release Notes shipped with your system.

The virtual memory must have been locked into physical memory by a call to **userdma**(D3X) or **useracc**(D3X). These functions also validate the user buffer. If the memory described by **b_un.b_addr** and **b_bcount** has not been validated and locked, the effects of **disjointio** are undefined and potentially catastrophic.

If the list of TA/TC pairs is to be used to control direct memory accessed (DMA) hardware, more work on the part of the caller is required. For example, it is typically necessary to add a null TA/TC pair to mark the end of the list. Some DMA devices require that bits be set in the upper part of each TC, while others require a transformation to another format, such as a linked list.

SEMAPHORE RAMIFICATIONS

No locks should be held when calling **disjointio**.

RETURN VALUE

If successful, **disjointio** returns the number of TA/TC pairs recorded in the disjoint array pointed to by *djntptr*. If not successful, **disjointio** returns −1, sets **b_error**, and sets **u.u_error** to the following:

ENXIO byte count of the I/O request exceeds the maximum allowed (determined by the kernel tunable parameter, DJNTMAXSZ), or more TA/TC pairs are required to describe the user virtual memory than are allowed by the *szdjnt* parameter.

disjointio also calls the **iodone**(D3X) function, unless the AIO flag in **b_flags** is set.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/disjointio.c

SEE ALSO

djntfree(D3X), **djntget**(D3X)

djntfree(D3X)

NAME

djntfree – free a disjoint I/O structure

SYNOPSIS

```
djntfree(entryp);
struct djntio *entryp;
```

ARGUMENTS

entryp disjoint I/O structure to be freed, as returned by **djntget**(D3X).

DESCRIPTION

djntfree frees a disjoint I/O that was allocated with **djntget**(D3X). Its argument is the value returned by **djntget**.

CAUTION **djntfree** *does not make any consistency checks.*

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **djntfree**.

RETURN VALUE

djntfree does not return a value under any condition.

LEVEL

Base or Interrupt

SOURCE FILE

io/disjointio.c

SEE ALSO

disjointio(D3X), **djntget**(D3X)

NAME

djntget – allocate a disjoint I/O data structure

SYNOPSIS

```
#include <sys/disjointio.h>
extern int djntesize;

struct djntio*;
djntget(slpflg);
int slpflg;
```

ARGUMENTS

slpflg indicates whether or not the process should block to await a disjoint I/O structure if one is not currently available. If set, the process will return NULL and not block if no disjoint I/O structure is available; if not set, the process will block until it can get a disjoint I/O structure.

DESCRIPTION

djntget returns a pointer to an array of disjoint I/O data structure. User virtual memory is typically discontiguous in physical memory. If the physical location of the virtual memory must be given to a routine, it can be described as a sequence of physical address / byte count pairs. Disjoint I/O data structures are used to hold such address/count sequences. The size of each disjoint I/O data structure array is given in the external variable DJNTESIZE. The value of DJNTESIZE determines the maximum size of a disjoint I/O data transfer and is determined by the tunable kernel parameter DJNTMAXSZ.[1]

The number of `djntio` structures available for use is limited; the actual number is determined by the **sysgen** parameter DJNTCNT. The structure should be freed back to the system pool using **djntfree**(D3X) when it is no longer required.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **djntget**.

RETURN VALUE

If successful, **djntget** returns a pointer to a `djntio` structure. The structure is actually the first in an array of structures. The size of the array is determined by the **sysgen** parameter DJNTMAXSZ and is given in the external variable DJNTESIZE.

[1] DJNTESIZE is determined by the following formula:

```
DJNTESIZE = ((NBPP-1+DJNTMAXSZ-1)/NBPP +1 +1)
```

NBPP, the number of bytes per page, is defined in *immu.h*. See *space.h* for more information about this calculation.

If no structure is available and *slpflg* is set, **djntget** returns NULL to the calling process.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/disjointio.c

SEE ALSO

disjointio(D3X), **djntfree**(D3X)

NAME

dma_breakup – set up **strategy** request using intermediate kernel buffering

SYNOPSIS

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

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

ARGUMENTS

strat address of a routine to be called, with a single parameter, a copy of the *bp* parameter to **dma_breakup**. Normally this routine will be the driver's **strategy**(D2X) routine.

bp pointer to a buf(D4X) structure

sectorsize sector size for data transfer

DESCRIPTION

On entry, the buf(D4X) structure pointed to by *bp* is assumed to be set up for a block device data transfer, except for the fact that the buffer address field points to an area of user virtual memory. This is the situation for subordinate functions called from **physio**(D3X).

The **dma_breakup** function provides a simple method of dealing with the fact that the buffer in virtual memory is possibly spread across discontiguous physical memory. It does this by providing a kernel buffer for the actual device transfer.

The *sectorsize* parameter is used to verify that the byte count specified in bp->b_bcount is for an integral number of sectors. If the byte count is correct, **dma_breakup** attempts to obtain a kernel buffer large enough to hold the entire transfer. If either of these tests fail, the **dma_breakup** routine sets an error condition, signals I/O completion (using the **iodone**(D3X) function) and returns.

dma_breakup determines the direction of transfer by the setting of the B_READ flag in the **b_flags** member of the buf structure pointed to by *bp*. For a write, data is copied from user space to the kernel buffer before the supplied *strat* routine is called. For a read, the *strat* routine is called and then data is copied from the kernel buffer. In both cases, **dma_breakup** blocks while waiting for the *strat* routine to signal completion with the **iodone** function.

dma_breakup blocks the driver with the **preiowait**(D3X) function; the actual **iowait**(D3X) function will be called at some other point within the operating system. Refer to **preiowait**(D3X) for a discussion of nested waits for I/O completion. The driver's interrupt routine must call **iodone**(D3X) to signal when the I/O transfer is completed.

In summary, **dma_breakup** requires the **b_flags**, **b_bcount**, and **b_un.b_addr** members in the supplied `buf`(D4X) structure. **dma_breakup** also requires the **u.u_drivsema** member in the `user`(D4X) structure to allow it to call the driver correctly.

On exit, **dma_breakup** may update the following members of `buf`:

`b_error` set to ENXIO if an error was encountered

b_flags The B_ERROR flag is set if an error was encountered. B_DONE and B_ERROR are explicitly cleared before the *strat* routine is called.

b_un.b_addr undefined. It was used to point to kernel buffer used for the transfer, but that memory may have been reused for another operation by the time **dma_breakup** exits.

Note that the *strat* routine will probably update additional `buf` members.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **dma_breakup**.

RETURN VALUE

No value is returned. If **dma_breakup** cannot allocate a buffer (typically because the transfer size exceeds the physical buffer size) or if the byte count specified in **`bp->b_bcount`** is not for an integral number of sectors, **b_flags** is ORed with B_ERROR and B_DONE and **b_error** is set to ENXIO.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/physdsk.c

SEE ALSO

strategy(D2X), **physio**(D3X), **userdma**(D3X)

EXAMPLE

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

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

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

/* physical read */

my_read(dev)
{
register int nblks;

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

my_write(dev)
{
register int nblks;

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

/*
 *  Ensure the request that came from physio will result in I/O to
 *  contiguous memory by using dma_breakup to obtain intermediate
 *  kernel buffering. Pass at least 512 bytes (one sector) at a
 *  time (except for the last request).
 */

static
my_breakup(bp)
register struct buf *bp;
{
  dma_breakup(my_strategy, bp, sectorsize);
}
```

NAME

driinvoke – fast lock on switch tables for driver semaphoring

SYNOPSIS

```
driinvoke(switch, major, minor, routine, parm);
```

ARGUMENTS

switch	identifies the switch table being accessed (`cdevsw` or `bdevsw`)
major	internal major device number entry
minor	internal minor device number entry
routine	name of entry point routine being accessed
parm	single parameter to *routine*

DESCRIPTION

The **driinvoke** macro is a faster alternative to **drilock**(D3X)/**driunlock**(D3X) that can be used when the invoked function is invoked with only a single parameter, and the return value from the function (if any) is ignored.

SEMAPHORE RAMIFICATIONS

driinvoke should be used only in fully-semaphored drivers. In drivers installed under the compatibility modes, **driinvoke**'s lock results in nested locks on the switch table entry, which causes reentry problems.

RETURN VALUE

None.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

sys/conf.h

SEE ALSO

drilock(D3X)/**driunlock**(D3X), `bdevsw`(D4X), `cdevsw`(D4X)

NAME

drilock, driunlock – lock switch table for semaphoring

SYNOPSIS

```
drilock(switch, major, minor)
cdevsw;               /* or bdevsw; */
driunlock(switch, major, minor)
int major;
int minor;
```

ARGUMENTS

switch identifies the switch table being accessed (cdevsw or bdevsw)

major internal major device number entry

minor internal minor device number entry

DESCRIPTION

The **drilock** and **driunlock** macros are used throughout the kernel to implement the device driver semaphoring policy by protecting calls to a driver through the switch tables. These are necessary for the REAL/IX Operating System because of the preemptive kernel and the multiprocessor configuration.

drilock behaves differently depending on the semaphoring policy under which the target driver is installed:

- ❏ For drivers installed as fully semaphored, **drilock** does nothing.
- ❏ For drivers installed under major- or minor-device semaphoring, **drilock** locks a semaphore, saving a pointer to it in **u.u_drivsema**.
- ❏ For drivers installed under CPU affinity, **drilock** does a context switch to the appropriate processor and disables preemption.

driunlock releases the semaphore and processes interrupts that may have been deferred while the driver semaphore was held.

Most drivers will not use these functions directly. A few drivers pass work on to other drivers by calling through the cdevsw table; these calls need to be protected by **drilock**.

SEMAPHORE RAMIFICATIONS

drilock and **driunlock** should be used only from fully-semaphored drivers. In drivers installed under the compatibility modes, **drilock**'s lock results in nested locks on the switch table entry, which causes reentry problems.

RETURN VALUE	None.
LEVEL	Base Only (Do not call from an interrupt routine)
SOURCE FILE	*sys/conf.h*
SEE ALSO	**driinvoke**(D3X), `bdevsw`(D4X), `cdevsw`(D4X), `user`(D4X)

NAME

ee_add, ee_rm – add to (remove from) a list of functions to be executed when the process exits or **execs**

SYNOPSIS

```
#include "sys/inline.h"
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/systm.h"
#include "sys/fs/s5dir.h"
#include "sys/signal.h"
#include "sys/immu.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/file.h"
#include "sys/inode.h"
#include "sys/fstyp.h"
#include "sys/region.h"
#include "sys/proc.h"
#include "sys/debug.h"
#include "sys/cmn_err.h"

int ee_add(func)
void (*func);

int ee_rm(func)
void (*func);
```

ARGUMENTS

func name of the function to be added to (removed from) a list of functions to be called when the process for which the driver invoked **ee_add** (**ee_rm**) exits or **execs** another process

DESCRIPTION

The **ee_add** and **ee_rm** kernel functions notify the driver when a critical process using that driver has exited or **execed**. **ee_add** should be used in a driver whenever sensitive information about the processes using the driver's services must be maintained to enable the driver to recover in the event that one of those processes suddenly goes away without properly cleaning up. **ee_rm** removes the argument *func* from the process' **exit/exec** function list. Interrupts should be enabled when **ee_add** or **ee_rm** is called.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

ee_add returns a 1 if the function *func* was already on the list or if it was successfully added; otherwise, it returns a 0 (zero).

ee_rm returns a 1 if the function *func* was successfully removed from the list; otherwise, it returns a 0.

LEVEL Base Only (Do not call from an interrupt routine)

SOURCE FILE *os/ee.c*

NAME

enable – reenable interrupts that were disabled with **disable**(D3X)

SYNOPSIS

`enable()`

ARGUMENTS

None.

DESCRIPTION

enable reenables interrupts that were disabled by **disable**(D3X). Refer to **disable**(D3X) for a discussion of when these functions are used rather than **spl*** or **spsema/svsema**.

386

CAUTION

On 386/486-based platforms, we strongly recommend using the **popsr***(D3X) and* **pushsrdisable***(D3X) kernel functions instead of the* **enable** *and* **disable***(D3X) functions, because* **enable** *does not restore the interrupt privilege level (ipl) but unconditionally sets it to zero.*

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

os//interrupt.c*

386

sys/inline.h

SEE ALSO

disable(D3X), **spsema**(D3X), **svsema**(D3X), **popsr**(D3X), **pushsrdisable**(D3X)

EXAMPLE

Refer to the example for **disable**(D3X).

freecpages(D3X)

NAME

freecpages – free contiguous pages previously allocated with **getcpages**

SYNOPSIS

```
freecpages(paddr, npgs)
unsigned int paddr, npgs;
```

ARGUMENTS

paddr physical address of the first in the range of contiguous pages to be freed (returned by **getcpages**(D3X). (This is returned by **getcpages**(D3X).

npgs number of pages in the range of contiguous pages.

DESCRIPTION

freecpages frees the set of contiguous pages previously allocated with **getcpages**. If a driver no longer needs the contiguous pages, it should free them. In many cases, the driver executes **getcpages** in its **init**(D2X) routine and never releases them.

The *npgs* is frequently expressed as:

`btoct(ctob(`*no_of_bytes*`)`

CAUTION

The number of pages freed must match the number of pages allocated with **getcpages**. *Freeing only part of the range of pages may corrupt the kernel.*

SEMAPHORE RAMIFICATIONS

No spin locks or global semaphores should be held when calling **freecpages**.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

os/page.c

SEE ALSO

getcpages(D3X)

freepbp(D3X)

NAME

freepbp – free buffer header obtained with **getpbp**(D3X)

SYNOPSIS

```
freepbp(bp)
buf_t* bp;
```

ARGUMENTS

bp pointer to the buffer header, returned by **getpbp**(D3X)

DESCRIPTION

freepbp frees the buffer header allocated with **getpbp**. **freepbp** places the buffer indicated by *bp* (which must have been allocated with **getpbp**) back on the free queue of physical buffer headers.

CAUTION

The kernel may be seriously corrupted if the values of the b_lock and b_iodone semaphores in the buf header are not the same when ***freepbp*** *is called as when* ***getpbp*** *was called. The values of the semaphores can change often, but must be returned to the original state before* ***freepbp*** *is called.*

The kernel may also be corrupted if ***freepbp*** *is called twice for the same allocation on the buffer.*

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **freepbp**.

RETURN VALUE

No value is returned.

LEVEL

Base or Interrupt

SOURCE FILE

os/physio.c

SEE ALSO

freephysbuf(D3X), **getpbp**(D3X), **getphysbuf**(D3X)

EXAMPLE

The following code illustrates how **freepbp** is used to free a buffer header:

```
if (ready_to_free_buffer_header) {
     freepbp(bp);
}
```

freephysbuf(D3X)

NAME

freephysbuf – release a physical buffer obtained with **getphysbuf**(D3X)

SYNOPSIS

```
freephysbuf(buffp)
caddr_t buffp;
```

ARGUMENTS

buffp pointer to physical buffer, returned by **getphysbuf**

DESCRIPTION

freephysbuf frees the physical buffer allocated by **getphysbuf** after the driver has finished with it (typically when an I/O transfer is complete). **freephysbuf** places the buffer indicated by *buffp* back on the queue of physical buffers.

CAUTION

The kernel may be corrupted if ***freephysbuf*** *is called twice for the same physical buffer.*

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **freephysbuf**.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

io/physdsk.c

SEE ALSO

getphysbuf(D3X)

EXAMPLE

The following code illustrates how **freephysbuf** is used to free a physical buffer when the I/O transfer is completed. *bufaddr* is the kernel buffer address. Refer to **getphysbuf**(D3X) for the associated code that allocated the physical buffer.

```
register caddr_t bufaddr;
bufaddr = getphysbuf(count);
if (I/O_complete) {
        freephysbuf(bufaddr)
}
```

NAME

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

SYNOPSIS

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

ARGUMENTS

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.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **fubyte**.

RETURN VALUE

The normal return value is the requested data bye. Otherwise, a –1 is returned if an attempt is made to access an address that is not part of a user program area.

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)

SOURCE FILE

ml//userio.s*

SEE ALSO

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

EXAMPLE

Refer to the **putc**(D3X) example for an example of how **fubyte** is called.

NAME

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

SYNOPSIS

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

ARGUMENTS

userbuf user program area address that contains a 32-bit word[1] 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.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **fuword**.

RETURN VALUE

The normal return value is the requested data word. Otherwise, a −1 is returned if an attempt is made to access an address that is not part of the user program area.

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)

SOURCE FILE

ml//userio.s*

SEE ALSO

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

EXAMPLE

When debugging a driver, the **ioctl**(D2X) routine can be used by the superuser to manually set a device control register. This can change any incorrect settings made by another driver routine.

- ❑ The driver verifies that the user-level process has real time or superuser privileges (line 19); if not, returns an error code (line 21). Note that `suser` sets the error code as a side effect.
- ❑ The new setting is retrieved from the user data area specified by *arg* (line 23).

[1] The **fushort** kernel function can be used to copy a 16-bit word. For **fushort**, *userbuf* must be short aligned.

- ☐ If *arg* is an invalid address, an error code is returned (line 26). Otherwise, the device control register is assigned the new setting (line 28).

```
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 (!(rtuser() || suser()))
20         {
21            return;
22         }
23         if ((c = fuword(arg)) == -1)
24         {
25            u.u_error = EFAULT;
26            return;
27         }
28         rp->control = c;
29         break;

30          ⋮
```

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;
```

ARGUMENTS

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).

Note that you must protect the `tty`(D4X) structure before manipulating it:

- ❑ If driver is installed under CPU affinity, set **splhi** to inhibit interrupts.
- ❑ If driver is installed under major- or minor-device semaphoring, issue a **psema**(D3X) against the semaphore you have initialized for the `tty`(D4X) structure.

SEMAPHORE RAMIFICATIONS

Drivers using **getc** must be installed under the compatibility modes.

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

SOURCE FILE

io/clist.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
getcb(D3X), **getcf**(D3X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **ttin**(D3X), **ttread**(D3X), `clist`(D4X)

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 areas and data in the clist, get a single byte from the first cblock in the clist (line 7),
- ❑ then copy it to the user data area (line 10).
- ❑ If an invalid address is found, then return error code (lines 11 – 12).
- ❑ Update remaining size of data area (line 14).

```
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){
7       if ((c = getc(&tp->t_canq)) == -1)
8          return;
9       }
10      if (subyte(u.u_base++, c) == -1){
11         u.u_error = EFAULT;
12         return;
13      }
14      u.u_count--;
15  }
```

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;
```

ARGUMENTS

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.

SEMAPHORE RAMIFICATIONS

Drivers that call **getcb** must be installed under the compatibility modes.

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

SOURCE FILE

io/clist.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
getcb(D3X), **getcf**(D3X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **ttin**(D3X), **ttread**(D3X), cblock(D4X)

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 (lines 7 through 9).
- If clist is empty, return (line 10).
- Next, compute the bytes in the cblock and copy the bytes to the user data area (lines 11 and 12).
- Finally, the empty cblock is returned to the cfreelist(D4X) (line 15).
- If an invalid address is detected, the data transfer returns an error condition (lines 16 and 17).

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

3     ⋮

4   register struct tty *tp = &xx_tty[minor(dev)];
5   register struct block *cp;
6   register int i;
7   while(u.u_count >= cfreelist.c_size)
8   {
9     if((cp = getcb(&tp->t_canq)) == NULL)  /* No cblocks available */
10        return;

11     i = cp->c_last - cp->c_first;
12     copyout (u.u_base, (caddr_t)&cp->c_data[cp->c_first],i);
13     u.u_base += i;                      /* Increment virtual base addr */

14     u.u_count -= i;                      /* Decrement bytes not transferred */
15     putcf(cp);                           /* Release cblock */

16     if (u.u_error != 0)
17         return;
18  }
```

NAME

getcf – get a free cblock(D4X)

SYNOPSIS

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

struct cblock *
getcf()
```

ARGUMENTS

None.

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.

SEMAPHORE RAMIFICATIONS

Drivers calling **getcf** must be installed under the compatibility modes.

RETURN VALUE

Under normal conditions, a pointer to a cblock is returned. Otherwise, if the cfreelist is empty, the system panics.

(Note that the initial number of cblocks in the system can be specified with the tunable parameter NCLIST. The system periodically checks the usage of cblocks and attempts to add more cblocks to the pool. Therefore, it is unlikely the system will ever run out of cblocks. Refer to cblock(D4X) for details.)

LEVEL

Base or Interrupt

SOURCE FILE

io/clist.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
getcb(D3X), **getcf**(D3X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **ttin**(D3X), **ttread**(D3X), cblock(D4X)

NAME

getcpages – get physically contiguous pages

SYNOPSIS

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

caddr_t
getcpages(npgs, mode)
int npgs;
unsigned mode;
```

ARGUMENTS

npgs number of contiguous memory pages (clicks) required

mode any of the following 32-bit flags; if MINADDRFLAG is used, *mode* also contains an address:

MINADDRFLAG
Used to specify the lowest area of physical memory from which the range of contiguous pages should be allocated. The address specified should be aligned on an even page boundary and is obtained by ANDing the mode parameter with the inverse of POFFMASK.

SETCI Used to specify that the allocated memory pages will be cache inhibited. The use of SETCI relies on the condition of the flag **badcache**. This flag is set in the kernel if hardware does not maintain cache coherency (e.g., as on the MVME187). Thus, SETCI can be specified only if **badcache** is set.

CAUTION *Specifying SETCI when badcache is not set causes the system to panic.*

NOSLEEP Do not block if physically contiguous pages cannot be allocated. Without this setting, the code will block and retry the page allocation a few times, although it will not necessarily block until it can allocate the pages.

CAUTION

If the NOSLEEP mode is not specified, **getcpages** *blocks for a period of time – in the order of seconds – while waiting for contiguous pages to become available. It is strongly recommended that* **getcpages** *without NOSLEEP should be used only during driver initialization and that NOSLEEP should be specified for all other calls to* **getcpages***.*

DESCRIPTION

getcpages gets a block of physically contiguous pages. Pages allocated are not mapped to **sysreg**. **getcpages** is commonly called from driver **init**(D2X) routines, and the range of contiguous pages is held as long as the system is running. If the range of contiguous pages is not required at all times, it can be freed with **freecpages**(D3X).

If **getcpages** is used any time after initialization, the number of available contiguous pages may be insufficient to satisfy the arguments to the call. If this happens, the process will hang and the following message will display:

```
getcpages--waiting for ## contiguous pages
```

where ## is the number of contiguous pages requested in the **getcpages** call.

SEMAPHORE RAMIFICATIONS

No locks and no global semaphores should be held when calling **getcpages** unless the NOSLEEP mode is specified.

RETURN VALUE

If successful, **getcpages** returns the kernel virtual address of the blocks allocated. If the pages cannot be allocated, **getcpages** returns 0.

LEVEL

Base or Interrupt. The NOSLEEP mode must be specified if calling from interrupt level.

SOURCE FILE

os/page.c

SEE ALSO

freecpages(D3X)

EXAMPLE

The following code illustrates how **getcpages** is used to allocate pages, specifying `0x100000` as the lowest address at which the pages can be allocated and not blocking if the pages cannot be allocated.

```
size = btoc(sum of all buffers to use the contiguous range of pages)

if (!(mblock = (mblk_t *)getcpages(size, NOSLEEP|MINADDRFLAG|0x100000)))
{
    cmn_err(CE_WARN, "myinit: cannot allocate contiguous pages");
    other error handling code
}
```

NAME

geteblk – get an empty block

SYNOPSIS

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

struct buf*
geteblk()
```

ARGUMENTS

None.

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. Buffers allocated with **geteblk** should be released with **brelse**(D3X) when they are no longer needed.

When the driver **strategy**(D2X) routine receives a buffer header from the kernel (that is, when the driver is entered through its **strategy** routine), 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_AGE to ensure that, when the buffer is released, it is placed at the head of the free queue and hence reused before other buffers that may contain valid data. If this buffer header is to be passed to any of the various kernel or driver routines, then certain other flags may be required to cause the required behavior. For example, if the buffer is passed to a block driver **strategy** routine, the B_READ flag must be set in order for a read to take place.

- ❑ **b_forw** and **b_back** are reserved for use by the buffer allocation routines and must not be altered by the driver.

- ❑ **b_avforw** and **b_avback** are undefined and available for use by the driver, typically for queuing the buffer.

- ❑ **b_dev** is set to NODEV and must be initialized by the driver.

- ❑ **b_error** is normally zero, but this is not guaranteed by the kernel. The normal usage of this field is to carry an error code. This field is checked for an error code only if the flag B_ERROR is set, in which case the error code is transferred to the **u.u_error** field of the user(D4X) structure, for eventual return to a caller. The field is cleared after **u.u_error** is set.

It is possible (but not recommended) for a driver to use this field for other uses. If it does do this, it should set the field to zero before releasing the buffer.

- □ **b_lock** will have had a successful **psema**(D3X) operation performed on it, indicating that the buffer is locked on behalf of its new owner. This semaphore is released by the operating system when the **brelse**(D3X) function frees the buffer header back to the free pool. Drivers should not perform any semaphore operations on this field other than the implicit **vsema**(D3X) operation when the buffer is released.

- □ **b_iodone** will have the value of 0 so that the first **psema** operation will block. The **iowait**(D3X) or **preiowait**(D3X) functions will issue a **psema** to block, and the **iodone**(D3X) function will issue a **vsema** operation to unblock; the driver should not perform an explicit semaphore operations on **b_iodone**.

- □ **b_bcount** is set to the number of bytes of data in the buffer pointed to by **b_un.b_addr**. **geteblk** returns a buffer of the smallest size configured in the system (usually 1 Kbyte).

- □ **b_un.b_addr** has been set to the kernel virtual address of the buffer that the buffer header is controlling. A driver should preserve this field because the kernel will assume it is valid when the driver issues the **brelse** function to release the buffer. If the buffer header is to be passed to the **dma_breakup**(D3X) function, take care because **dma_breakup** will overwrite the value of this field.

- □ **b_resid** member will be set to zero. This field is conventionally used to carry the residual byte count if not all the requested data is transferred. The zero value means that **b_resid** is preset for the case where a complete transfer takes place.

- □ **b_shift** is reserved for use by the buffer header allocate and search routines; it should not be read or written by the driver.

- □ The **b_s0, b_s1, b_s2, b_umd, b_blkno, b_start**, and **b_proc** members are undefined.

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, because 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. Note, however, that if a driver wishes to obtain a buffer header structure that is not associated with any particular buffer, then it may use the **getpbp**(D3X) function.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **geteblk**.

RETURN VALUE

A pointer to a buf(D4X) structure is returned.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/bio.c

SEE ALSO

KPG, "Synchronized I/O Operations"
strategy(D2X), **brelse**(D3X), **dma_breakup**(D3X), **getnblk**(D3X), **getpbp**(D3X), **iowait**(D3X), **iodone**(D3X), **preiowait**(D3X), buf(D4X)

EXAMPLE

The example given for **brelse**(D3X) illustrates the use of **geteblk**.

NAME

getnblk – get empty buffer of specified size

SYNOPSIS

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

struct buf*
getnblk(bf, need)
bfree_t *bf;
int need;
```

ARGUMENTS

bf pointer to the free list holding buffers of the desired size. The *sys/buf.h* file declares an array of lists named **bfree**. The elements determine the buffer size being requested.[1] For example:

bfree[0]	controls 1-Kbyte buffers
bfree[1]	controls 2-Kbyte buffers
bfree[2]	controls 4-Kbyte buffers
bfree[3]	controls 8-Kbyte buffers
bfree[8]	controls 128-Kbyte buffers

need determines the response if no buffer can be allocated. If set to 1, the system will panic if a buffer cannot be allocated; if set to 0, **getnblk** returns NULL if a buffer cannot be allocated.

DESCRIPTION

The **getnblk** function gets an empty buffer that is at least as big as that in the freelist pointed to by *bf*. The system must be configured with buffers at least as large as that specified. The state of the returned buffer is the same as that described for **geteblk**(D3X).

SEMAPHORE RAMIFICATIONS

No semaphores should be held when calling **getnblk**.

RETURN VALUE

If successful, **getnblk** returns the buffer pointer for the allocated buffer. If not successful, the *need* argument determines the outcome:

- ❑ If *need* is 1 and no buffer can be allocated, the system panics and gives the following error message: "**getnblk: no *size* byte buffers**".
- ❑ If *need* is 0, **getnblk** returns 0; the driver or system call should take appropriate action, which may include setting the **u.u_error** member of

[1] The specified buffer size must be configured as part of the system buffer cache. The REAL/IX Operating System supports buffer sizes ranging from 1 Kbyte to 128 Kbytes, but the released configuration uses only 1 Kbyte. Refer to the *System Administrator's Guide* for more information.

the user(D4X) structure to ENOMEM or some other value agreed on between the system call and the user-level process (it is not necessary to set **u.u_error**; this is determined by the needs of the application).

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/bio.c

SEE ALSO

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

EXAMPLE

The following code illustrates how **getnblk** is used to obtain a buf(D4X) with an associated buffer whose size is 4 Kbytes:

```
if (getting_the_buffer_is_essential) {
     mp->m_bufp = (caddr_t)getnblk(&bfree[2], 1);
} else {
     qp->q_bufp = (caddr_t)getnblk(&bfree[2], 0);
     if (qp->q_bufp == 0) {
          u.u_error = ENOMEM;
          return;
     }
}
```

qp->q_bufp == 0 is true if no buffer is obtained.

NAME

getpbp – get physical I/O buffer pointer

SYNOPSIS

```
buf_t *
getpbp(slpflg)
int slpflg;
```

ARGUMENTS

slpflg indicates whether or not the process should block to await a physical I/O pointer if one is not currently available. If set, the process will return NULL and not block if no physical I/O pointer is available; if not set, the process will block until it can get a physical I/O buffer pointer.

DESCRIPTION

getpbp obtains a buffer header structure for use in making calls to block mode routines that bypass the buffer cache.

The contents of the buf structure returned by **getpbp** are undefined except that the semaphores **b_lock** and **b_iodone** are correctly initialized to values of 1 and 0, respectively. After the I/O operation is complete, the driver should return the buf to the poll of physical buffer headers with the **freepbp**(D3X) function.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **getpbp**.

RETURN VALUE

If successful, **getpbp** returns the buffer pointer for the physical I/O buffer. Otherwise, it returns a null pointer.

If *slpflg* is set and no buffer pointer is returned, the action to be taken is driver dependent. If running at base level and the initiating operation cannot be accomplished due to lack of resources, it is usually appropriate to set the **u.u_error** member of the user(D4X) structure to EAGAIN.

LEVEL

Base or Interrupt; if called from interrupt level, *slpflg* must be set.

SOURCE FILE

os/physio.c

SEE ALSO

freepbp(D3X), **physck**(D3X), **physio**(D3X)

EXAMPLE

The following code segment illustrates how **getpbp** is used:

```
#define NOSLP 1

  :

if ( (bp = getpbp(NOSLP)) == NULL ) {
     cmn_err(CE_WARN,"unable to allocate buffer header");
     u.u_error = EAGAIN;
     return;
}
:
```

NAME

getphysbuf – get physical buffer

SYNOPSIS

```
caddr_t
getphysbuf(size)
unsigned size;
```

ARGUMENTS

size specifies the minimum buffer size required

DESCRIPTION

getphysbuf obtains a physical buffer, which is an area of kernel memory typically used as an intermediate buffer between user virtual memory and a device driver.[1]

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **getphysbuf**. The function sets a spin lock on the linked list of physical buffers, then releases it after it has obtained the buffer. Because **getphysbuf** may block until a buffer is obtained, semaphores should be used with caution.

RETURN VALUE

If successful, **getphysbuf** returns a pointer to a buffer that is guaranteed to be greater than or equal to the specified size. If *size* is greater than the configured PHYBSIZE, it returns a NULL pointer.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/physdsk.c

SEE ALSO

freephysbuf(D3X), **getpbp**(D3X), **freebpb**(D3X)

[1] The number of physical buffers configured in the system and the size of each are determined by the PHYSCNT and PHYBSIZE tunable parameters discussed in the *System Administrator's Guide*.

EXAMPLE

The following code illustrates how **getphysbuf** is used to obtain a physical buffer. Refer to **freephysbuf**(D3X) for the associated code that frees this physical buffer after the I/O transfer is complete.

```
register caddr_t bufaddr;
register int count

count = bp->b_bcount
if ((bufaddr = getphysbuf(count)) == 0) {
      bp->b_flags |= B_ERROR;
      bp->b_error = ENXIO;
      iodone(pb);
      return;
}
```

NAME

get_timer – get interval timer

SYNOPSIS

```
struct tmr *get_timer(type);
int type;
```

ARGUMENTS

type the timer type to be used by this interval timer; at present, valid values are TIMEOFDAY and TIMESINCEBOOT

DESCRIPTION

The **get_timer** function acquires an interval timer from the pool of available interval timers. The resource is then allocated uniquely to the driver that issued **get_timer** until the driver releases the timer by issuing **rel_timer**(D3X). When used with **get_timer**, TIMESINCEBOOT gives the same results as TIMEOFDAY.

A successful call to **get_timer** actually returns a pointer to the tmr structure. This structure is defined in *sys/timesys.h*. Note, however, that the contents of this structure may change, so drivers should not use any of the fields within the tmr structure.

If no interval timers are available system-wide or if none are available for system use (as determined by the tunable parameters ITIMAXSYS and ITIMAXK, respectively), **get_timer** returns NULL.[1] **get_timer** also returns NULL if *type* is not a valid timer type or if the timer type supports only a limited number of timers and the limit has already been reached.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

If successful, **get_timer** returns a pointer to the tmr structure allocated to the driver. **get_timer** returns NULL under any of the following conditions:

- ❑ no interval timers are available
- ❑ *type* is not a valid timer type
- ❑ the number of timers supported by *type* has already been reached

[1] Three other tunable parameters that control interval timers are ITIMAXPROC, which limits the number of processes that can have timers at any time; ITICNTPROC, which determines how many interval timers a process can have; and CLOCKRES, which sets the system clock rates and allows for adjustment of the clock resolution. For more information about these parameters, refer to the *System Administrator's Guide*.

LEVEL	Base Only (Do not call from an interrupt routine)
SOURCE FILE	*os/timer.c*
SEE ALSO	**rel_timer**(D3X), **set_timer**(D3X)

NAME

386

inb, inw, inl – read a specific 80x86 I/O address (port)

SYNOPSIS

```
inb(port)
int port;
```

The synopses of **inw** and **inl** are the same as the synopsis of **inb**.

ARGUMENT

port	the address to be read

DESCRIPTION

The function **inb**, **inw**, or **inl** reads a byte, a short (16-bit) value, or a long (32-bit) value, respectively, in the 80x86 I/O address space.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

Value read at the I/O address.

LEVEL

Base or Interrupt

SOURCE FILE

sys/inline.h

SEE ALSO

out(D3X)

NAME

incsema, rincsema, pincsema – increment a semaphore value for a resource by 1

SYNOPSIS

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

incsema(sem_addr)
sema_t *sem_addr;
```

The synopses of **rincsema** and **pincsema** are the same as that of **incsema**.

ARGUMENTS

sem_addr identifies the semaphore to be incremented

DESCRIPTION

The **incsema** family of macros increment by one the value of the semaphore specified by *sem_addr*. They are used to manipulate counters (such as the number of I/O operations in progress) for statistics, and should not be used for synchronization or exclusion.

rincsema and **pincsema** provide functionality similar to that of **incsema**, but are faster. **rincsema** can be used when all interrupts are disabled with a spin lock; **pincsema** can be used when all interrupts are guaranteed to be enabled.

SEMAPHORE RAMIFICATIONS

Drivers that call **incsema** should be installed fully semaphored.

RETURN VALUE

The **incsema** macros do not return a value under any conditions.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sema.h

SEE ALSO

decsema(D3X)

NAME

initlock – initialize spin lock for a resource

SYNOPSIS

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

initlock(lock_addr, lock_val)
lock_t *lock_addr;
int lock_val;
```

ARGUMENTS

lock_addr identifies the spin lock to be initialized; this addr is used by the macros that set and release the spin lock.

lock_val the value to which the semaphore is to be initialized. If 0, the semaphore is initially unlocked; if 1, the semaphore is initially locked. Other values are illegal.

DESCRIPTION

The **initlock** function is used in the driver's **init**(D2X) routine to initialize the spin lock for a resource to 0 (unlocked) or 1 (locked). The predominant usage is to initialize a spin lock to be unlocked (*lock_val* = 0).

The number of locks that need to be initialized varies from driver to driver. Some drivers are served well by one global lock that is used for all spin operations, whereas other drivers require as many as one lock per board or per minor device. The spinning action involved when a process is attempting to access a locked spin lock hurts system performance as well as the performance of the driver itself. Therefore, for performance, it is best to be generous in the number of spin locks initialized. Spin locks also disable interrupts for the CPU; for this reason, they should be locked for only very short periods of time (typically less than 50 microseconds).

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

sys/sema.h

SEE ALSO

KPG, "Synchronization"
spsema(D3X), **svsema**(D3X), **valulock**(D3X)

EXAMPLE

```
#include    "sys/debug.h"
#include    "sys/sema.h"

extern struct xyz    xyz_tab[];          /* xyz table */
extern struct xx     xx;                 /* information structure */

xx_init()

{
   register int   i;

   for (i = 0; i < xx.maxsys; i++){     /* initialize all locks */
        xyz_tab[i].z_key = Z_FREE;
        xyz_tab[i].z_cid = i;
        initlock(&xyz_tab[i].z_lock, 0);
   }
}
```

NAME

initsema, reinitsema, rreinitsema, preinitsema – initialize or reinitialize kernel semaphore for a resource

SYNOPSIS

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

initsema(sem_addr, sem_val, flags);
sema_t *sem_addr;
int sem_val;
int flags;
```

The synopses of the **reinitsema, rreinitsema** and **preinitsema** macros are the same as that for **initsema.**

ARGUMENTS

sem_addr identifies the semaphore to be initialized; this address is used by the services that lock and unlock semaphores.

sem_val the value to which the semaphore is to be initialized. If 1, the semaphore is initially unlocked; if 0, the semaphore is initially locked. If greater than 1, signifies the number of processes that can concurrently access the resource. Negative values are illegal.

flags currently unused; must be specified as 0.

DESCRIPTION

The **initsema** function is used in the driver's **init**(D2X) routine to initialize the semaphore for a resource to a non-negative integer value. The value of *sem_val* determines the type of access for the resource:

0 the semaphore for the resource is initially locked and waits for an unlock operation. For instance, a process can wait for completion of an I/O operation when *sem_val* is 0. A call to **psema**(D3X) will block the calling process until a **vsema**(D3X) is issued against the resource when the I/O operation is complete.

1 sets up mutual exclusion access; allows only one process to access the resource at a time. For instance, a critical section of code can be protected when *sem_val* is 1. The first process to access the critical section of code with **psema** will be successful, but the next process that attempts to access the same section of code will block waiting for a **vsema**, which will allow access to that section of code.

>1 a specified number of processes can concurrently access the resource. For instance, if *sem_val* is 3, the first three processes that access the resource with **psema** or **cpsema**(D3X) will be successful, but the fourth process will block waiting for a **vsema**, which will allow access to the resource.

The **reinitsema** macro reinitializes a semaphore that was previously initialized with **initsema**. It is used, for example, to ensure that a semaphore used for waiting for I/O completion has a value of 0 before a process issues a **psema** call that should block the process.

The **rreinitsema** and **preinitsema** macros are faster than **reinitsema**; they can be used to optimize performance. **rreinitsema** can be used if interrupts have been disabled; **preinitsema** can be used if all interrupts are guaranteed to be enabled.

Note that the **reinitsema** semaphore family are rarely used because the **psema** and **vsema** operations normally ensure a semaphore has the required valued.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

The **initsema** macros do not return a value under any conditions.

LEVEL

initsema(D3X) – Base Only (Do not call from an interrupt routine)
reinitsema(D3X) – Base or Interrupt

SOURCE FILE

os/sema.c

SEE ALSO

KPG, "Synchronization"
cpsema(D3X), **cvsema**(D3X), **decsema**(D3X), **incsema**(D3X), **psema**(D3X), **valulock**(D3X), **valusema**(D3X), **vsema**(D3X)

EXAMPLE

As an aid to understanding how to use the **initsema** macros, refer to **psema**(D3X).

In this example, **initsema** is initializing two semaphores for a pool of buffers. The first lock is for individual buffers; the buffers are allocated to a process one-at-a-time, and no lock is required as long as there are available buffers.

A second semaphore, for the download buffer itself, is initialized to 1. It is used in the **lock_dlbuf** and **unlock_dlbuf** routines to control access to the buffer resource. Note how **lock_dlbuf** uses the **psema** routine to check for pending signals before blocking. If there are pending signals, it records an error condition to the `user` structure and does a **klongjmp**; otherwise, it blocks and waits for the **unlock_dlbuf** routine to release the semaphore.

```
xx_init                                        /* initializes buffer semaphores */
   for ctl = 0; ctl < xx_cnt/MAXDEV); ctl++) {
      initsema(&de[ctl].freesema, NPKTS-2, 0);
      initsema(&de[ctl].buf_busy, 1, 0);   /* lock for download buffer */
   }

lock_dlbuf(dp)                                  /* lock download buffer */
register struct xx_dev *dp;
{
   if (psema(&dp->buf_busy, SEMCATCH)) {    /* has the psema been */
      u.u_error = EINTR;                   /* interrupted by a signal? */
      klongjmp(u.u_qsav);
   }
}

unlock_dlbuf(dp)                                 /* unlock download buffer */
register struct xx_dev *dp;
{
   vsema(&dp->buf_busy, 0, 0);
}
```

io_alloc(D3X)

NAME

MBII

io_alloc – allocates virtual memory-mapped I/O address space

SYNOPSIS

```
io_alloc(paddr, len)
unsigned int paddr;
int len;
```

ARGUMENTS

paddr physical address to which virtual I/O memory is mapped

len number of bytes of contiguous virtual I/O memory to be allocated

DESCRIPTION

io_alloc dynamically allocates *len* bytes of contiguous virtual I/O memory and maps it to physical addresses starting at *paddr*. Allocation is always made in page size granularity.

io_alloc should be used only in the *xxx***init**(D2X) function of a driver.

SEMAPHORE RAMIFICATIONS

None

RETURN VALUE

If the call to **io_alloc** is successful, it returns the starting address of the virtual memory-mapped I/O space; otherwise, if not enough free memory is available, it returns −1.

LEVEL

Base or Interrupt

SOURCE FILE

io/mbus/mb2cpu.c

NAME

iodone – resume execution suspended pending block I/O

SYNOPSIS

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

iodone(bp)
struct buf *bp
```

ARGUMENTS

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 (or should have occurred).

DESCRIPTION

iodone is normally called by the block driver interrupt routine when the data transfer is complete. It is also called if an error condition prevents the completion of the data transfer. **iodone** does the following:

- ❑ Marks **b_flags** of buffer header with B_DONE.
- ❑ If the I/O operation is synchronous, issues a **vsema**(D3X) to unblock a process that called **iowait**(D3X) to wait for the buffer header.
- ❑ If the I/O operation is asynchronous, releases the buffer (**brelse**(D3X))

SEMAPHORE RAMIFICATIONS

No spin locks should be set when calling **iodone**.

RETURN VALUE

Under all conditions, no value is returned.

LEVEL

Base or Interrupt

SOURCE FILE

os/bio.c

SEE ALSO

KPG, "Synchronization"
iowait(D3X), **preiowait**(D3X), **psema**(D3X), **sleep**(D3X), **vsema**(D3X), **wakeup**(D3X), buf(D4X)

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 (line 10). The return value for the **read**(2) system call is computed by taking the difference between **b_bcount** and **b_resid**.
- ❑ Otherwise, if the request is outside the limits of the device, the routine will report an error condition (lines 11 through 14).
- ❑ In either case, report the I/O operation as complete and (line 15). **iodone** unblocks the process that is blocked waiting for this I/O operation or, if this is an asynchronous I/O operation (B_ASYNC), releases the buffer.

```
 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) {
 9          if (blkno == RAMDNBLK && bp->b_flags & B_READ) {
10                  bp->b_resid = bp->b_bcount;

11          } else {
12                  bp->b_error = ENXIO;
13                  bp->b_flags |= B_ERROR;
14          }

15         iodone(bp)
16         return;
17   }
18   /* continue to set up transfer */
```

NAME

iomove – move bytes

SYNOPSIS

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

ARGUMENTS

cp bytes are moved to or from this address in kernel space.

bytes number of bytes to move. If *bytes* is 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 user address space to a driver. Set to B_READ to move bytes from a driver to user address space.

DESCRIPTION

This function copies bytes from user space to a driver, or from a driver to a user space. The kernel address is given by the *cp* parameter, while the user address is given by the **u.u_base** field of the user(D4X) structure. The **u.u_segflg** (described in *user.h*) determines how the copy is made. If **u.u_segflg** shows that this is a kernel process (segflag==1), then a straightforward copy can be made; otherwise, virtual address translations must be made.

iomove cannot be called from the driver's **init**(D2X) 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.

SEMAPHORE RAMIFICATIONS

No spin locks should be set when calling **iomove**.

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)

SOURCE FILE *os/move.c*

SEE ALSO *KPG*, "Synchronized I/O Operations"
bcopy(D3X), **copyin**(D3X), **copyout**(D3X), **fubyte**(D3X), **fuword**(D3X), **subyte**(D3X), **suword**(D3X)

EXAMPLE With a RAM disk, direct I/O requests can be handled in the driver's **read**(D2X) routine (begins line 4) and **write**(D2X) routine (begins line 24), 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 (lines 12 and 32) to determine if the I/O request is in the limits of the RAM disk (**physck**(D3X)).
- □ The number of blocks the user data area can contain is computed (lines 14 and 34). The data must be moved as a single complete block or multiples of complete blocks, so the user data area must be large enough to contain at least one complete block. If it cannot, an error condition will be returned for read operations (line 17), or must be set for write operations (line 36).
- □ Otherwise, compute the starting block number (lines 19 and 39) and copy the requested number of blocks from the RAM disk to the user data area (lines 20 and 40).

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

ramdread(dev)
dev_t dev;
{
register daddr_t blkno;              /* Starting block number */
register int   nblocks;              /* # blocks to be read with physio */
   if (physck(RAMDNBLK,B_READ)) {
           if ((nblks = u.u_count / RAMDBSIZ)) <= 0)
                   return;
   }                                 /* endif */
   blkno = u.u_offset /RAMDBSIZ;
   iomove(&ramdblks [blkno][0], (nblks * RAMDBSIZ), B_READ);
      /* Copy data to user */

}                                    /* end ramdread */
ramdwrite(dev)
dev_t dev;
{
register daddr_t blkno;           /* Starting block number */
register int nblks;               /* # blocks to be written with physio */
   if (physck(RAMDNBLK,B_WRITE)) {
            if (u.u_count % RAMDBSIZ !=0 )) {
                    u.u_error = EFAULT;
                    return;
            }                        /* endif */
            blkno = u.u_offset / RAMDBSIZ;
            iomove(&ramdblks[blkno][0], u.u_count, B_WRITE);
   }
}                                    /* end ramdwrite */
```

NAME

iowait – block 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;
```

ARGUMENTS

bp	pointer to a buf(D4X) structure controlling the data transfer

DESCRIPTION

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

iowait blocks on the **b_iodone** semaphore to wait for I/O completion. The semaphore is unblocked by a corresponding call to **iodone**(D3X) when the transfer completes.

Do not call **iowait** from the driver **init**(D2X), **strategy**(D2X), or interrupt routine. When you need **iowait** functionality in the **strategy** routine or when using **physio**(D3X), use the **preiowait**(D3X) function instead. Refer to **preiowait**(D3X) for details.

SEMAPHORE RAMIFICATIONS

No spin locks can be set when calling **iowait**.

RETURN VALUE

No value is returned.

This function updates **u.u_error** with information in **b_error** on errors that occurred while the process was blocked. 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)

SOURCE FILE

os/bio.c

SEE ALSO

iodone(D3X), **psema**(D3X), **preiowait**(D3X), **sleep**(D3X), **vsema**(D3X), **wakeup**(D3X)

EXAMPLE

Refer to the **geteblk**(D3X) example for an example of using **iowait**(D3X).

NAME

klongjmp - non-local "goto"

SYNOPSIS

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

void klongjmp();
```

ARGUMENTS

None.

DESCRIPTION

This function restores a previously saved environment, then transfers control to this environment.

By default, the restored environment is that of the system call handler. In this case, the system call handler ensures that an error return is made from the system call. If no error code is set in **u.u_error**, **klongjmp** sets EINTR.

You can set an alternative return environment by using the **ksetjmp**(D3X) function. **klongjmp** returns control to this alternative environment if the **u.u_setjmp** flag is set. In this case, **klongjmp** ensures that **u.u_setjmp** is cleared, but does not check **u.u_error** to see if EINTR should be set.

klongjmp is rarely called explicitly by a driver. However, you should be aware that it is called when a process is interrupted while sleeping on an interruptible semaphore. For more information, refer to **psema**(D3X) and **sleep**(D3X).

klongjmp is the equivalent of the UNIX System V **longjmp** kernel function. This function is a part of the kernel. It is *not* the same as the **longjmp** library routine (part of the **setjmp**(3C) routine). Both the code and the number of arguments are different.

klongjmp is useful when your code has entered many successive layers of subroutines and you wish to return immediately to an upper level. 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. **klongjmp** provides a quick return to the user program that issued the call to the driver.

When a blocking system call is terminated prematurely by a signal, it is necessary to abort the system call in an orderly manner before returning to the calling process. **klongjmp** provides a convenient method of doing this.

Drivers that block may need to perform cleanup operations before **klongjmp** is called. Typical items that need cleaning up are locked data structures that should be unlocked when the system call is finished. If the SEMCATCH flag is specified for **psema** (or the **sleep** priority argument is ORed with the defined constant PCATCH), **klongjmp** is not called when a signal is received; instead, the value 1 is returned to the calling routine, and the driver must call **klongjmp** explicitly after doing the necessary cleanup.

A default return environment is set up at the beginning of every system call. Therefore, a driver can always use **klongjmp** to abandon normal processing when an error is detected in the base level.

Note that interrupts should be enabled when **klongjmp** is called; that is, it is the caller's responsibility to enable interrupts.

When you set an alternative environment to be restored (by setting **u.u_setjmp** and calling **ksetjmp**), the environment details are stored in the fixed area **u.u_qsav**. Therefore, it is not possible to stack return environments. If it is necessary to arrange for a temporary alternative return environment, an explicit save area can be given to the **osetjmp**(D3X) function, and control can be returned to that save area by a call to **olongjmp**(D3X). In practice, **osetjmp** and **olongjmp** are rarely used.

SEMAPHORE RAMIFICATIONS

No spin locks should be set when calling **klongjmp**.

RETURN VALUE

None (Because this function performs a non-local "goto", it does not return to the caller)

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

ml//cswitch.s*

SEE ALSO

psema(D3X), **sleep**(D3X)

EXAMPLES

Fully Semaphored

Any code that blocks with the SEMINTR flag (or a flag that implies SEMINTR) set can have the I/O request aborted upon receiving any signal. Control returns to the appropriate location. 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 SEMCATCH flag when **psema** is called. If the return code value from **psema** is −1, then the **vsema** results from receiving a signal.
- ❑ do the necessary cleanup code and call **klongjmp** to return control to the appropriate location.

```
if (psema(this_sema,SEMCATCH) == -1 {
     do whatever cleanup is necessary
     u.u_error = EINTR;
     klongjmp();
}
```

Compatibility Modes

Drivers installed under the compatibility modes issue **sleep**(D3X) with a priority greater than PZERO (defined in *param.h*) to make the **sleep** interruptible. To "catch" the interrupt and do cleanup before returning with a call to **klongjmp**:

- ❑ OR the PCATCH bit is to the value in the priority field. In the example, this is done by defining XX_PRIORITY in the first line.
- ❑ If the return code value from **sleep** is equal to 1, then the **wakeup** results from receiving a signal.
- ❑ do the necessary cleanup code and call **klongjmp** to return control to the appropriate location.

```
#define XX_PRIORITY ((PZERO + 1) | PCATCH)

     if (sleep(&event, XX_PRIORITY)==1) {
          do whatever cleanup is necessary
          u.u_error = EINTR;
          klongjmp();
     }
```

NAME

kmap – lock user virtual memory and map it to kernel virtual memory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/errno.h
#include <sys/systm.h>

caddr_t
kmap(base, count, flags);
caddr_t base;
int count;
int flags;
```

ARGUMENTS

base the start address of the user memory to be mapped

count the size in bytes of the user memory to be mapped

flags valid *flags* values are:

B_READ map the address space for user read operations (i.e., the kernel may later try to write to it); may be ORed with B_PHYS

B_WRITE map the address space for user write operations (i.e., the kernel may later try to read from it); may be ORed with B_PHYS

B_PHYS do not lock the pages into memory; normally used if the caller has already locked the pages in (most likely with the kernel macro **klock()** defined in *sys/klock.h*); OR with B_READ or B_WRITE

DESCRIPTION

kmap is typically used when the kernel (which includes the various drivers) may require access to an area of user memory when the user process is not currently executing.

The effect of **kmap** is undone by **kunmap**(D3X).

kmap checks that the user has access to the region of memory; there is no need to check this with **useracc**(D3X) before calling **kmap**.

SEMAPHORE RAMIFICATIONS

No spin locks should be set when calling **kmap**.

RETURN VALUE

If successful, the return value will be a pointer to the area of kernel virtual memory where the user virtual memory has been mapped. If unsuccessful, a null pointer is returned and **u.u_error** will be set with an appropriate error code:

EAGAIN	Insufficient kernel resources to lock or map a page
EFAULT	User memory is marked as being read-only. (A read from a device has to write to user memory, and it is not allowed.)
EFAULT	The memory described by *base* and *count* is not within the user's address space.
EINVAL	The count parameter was equal to zero.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/kmap.c

SEE ALSO

kunmap(D3X), **undma**(D3X), **userdma**(D3X)

NAME

ksetjmp – saves registers and return location for **klongjmp**(D3X) to **u.u_qsav**

SYNOPSIS

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

u.u_setjmp = 1;
int ksetjmp()
   ⋮
u.u_setjmp = 0
```

ARGUMENTS

None.

DESCRIPTION

ksetjmp sets the return value for future implicit and explicit calls to **klongjmp**(D3X) so that, if a signal is received or an error occurs, control can be returned to a specific section of code. Note that the default environment to which **klongjmp** returns is the system call handler; because this environment is suitable for most handlers, **ksetjmp** is rarely used.

ksetjmp returns the value zero (0) after saving environment details. If a call to **klongjmp** returns control to this point, it will appear as if the corresponding call to **ksetjmp** had just returned the value 1.

ksetjmp saves environment details in **u.u_qsav**. The calling process must set **u.u_setjmp** to indicate that the contents of **u.u_qsav** are valid and must clear **u.u_qsav** when a return to the environment saved in **u.u_qsav** is no longer required.

If **ksetjmp** is called a second time, it overwrites the previously saved environment in **u.u_qsav**. If it is necessary to stack return environments, use **osetjmp**(D3X) and **olongjmp**(D3X).

SEMAPHORE RAMIFICATIONS

No semaphores should be set when calling **ksetjmp**.

RETURN VALUE

0 if a normal call to **ksetjmp**. 1 if control has been returned to **ksetjmp** by a **klongjmp** call.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

ml//cswitch.s*

SEE ALSO

klongjmp(D3X), **olongjmp**(D3X), **osetjmp**(D3X)

EXAMPLE

The following code from the kernel **copen** function illustrates the use of **ksetjmp**. Note the use of **setjmpcleanup**. This function is called by kernel code (however, *not* by driver code) to clean up after every call to a driver; it is used in the event the driver that was called is configured under one of the compatibility modes.

```
u.u_setjmp = 1;
if (ksetjmp())  {
     setjmpcleanup();
     if (u.u_error == 0)
          u.u_error = EINTR;
     u.u_ofile[i] = NULL;
     closef(fp);
} else {

      ⋮

     u.u_setjmp = 0;
}
```

NAME

kunmap – unmap and unlock user virtual memory from kernel virtual memory

SYNOPSIS

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

void
kunmap(base, count, kvaddr, flags);
caddr_t base;
int count;
caddr_t kvaddr;
int flags;
```

ARGUMENTS

base The start address of the user memory to be unmapped.

count The size in bytes of the user memory to be unmapped.

kvaddr The start address of the kernel memory to which the user memory was mapped, as returned from an earlier call to **kmap**(D3X).

flags Must be the same as the *flag* argument specified in the corresponding call to **kmap**; valid *flags* values are:

B_READ the area was mapped for user read operations (i.e., B_READ was specified for **kmap**); may be ORed with B_PHYS

B_WRITE the area was mapped for user write operations (i.e., B_WRITE was specified for **kmap**); may be ORed with B_PHYS

B_PHYS do not unlock the pages from memory; typically used if the caller will unlock the pages later (most likely with the kernel macro **kunlock**() defined in *sys/klock.h*); OR with B_READ or B_WRITE

DESCRIPTION

kunmap is the inverse of **kmap**(D3X).

CAUTION

***kunmap** assumes that the parameters it is given are exactly as per the original call to **kmap**. In any case, it has no ready means by which to validate them. Passing incorrect parameters to the **kunmap** function will give undefined and potentially catastrophic results.*

SEMAPHORE RAMIFICATIONS

No spin locks should be set when calling **kunmap**.

RETURN VALUE

kunmap does not return a value.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/kmap.c

SEE ALSO

kmap(D3X)

NAME

major – return the internal major number from a device number

SYNOPSIS

```
int
major(dev)
dev_t dev;
```

ARGUMENTS

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. Installing your driver through the *custom.mk* file automatically provides –DINKERNEL.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

The internal major number.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sysmacros.h

SEE ALSO

makedev(D3X), **minor**(D3X)

EXAMPLE

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

NAME

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

SYNOPSIS

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

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. Typically, a defined constant is used to represent the major number used by device drivers.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

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

LEVEL

Base or Interrupt

SOURCE FILE

sys/sysmacros.h

SEE ALSO

major(D3X), **minor**(D3X)

NAME

malloc – allocate space from a private space management map

SYNOPSIS

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

uint
malloc(mp, size, 0)
register struct map *mp;
register int size;
```

ARGUMENTS

mp	memory map from which the resource is drawn
size	number of units of the resource
0	always 0 for drivers; **malloc** used outside drivers occasionally uses other values

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. The map should be protected by a semaphore defined in *map.h*. When accessing an internal memory map in a fully-semaphored driver, **malloc** locks the semaphore before doing the allocation, then frees it.

SEMAPHORE RAMIFICATIONS

A semaphore is set automatically when **malloc** is called if a semaphore was specified in the previous call to **mapinit**(D3X).

RETURN VALUE

Under normal conditions, **malloc** returns the address of the buffer (as an unsigned integer). Otherwise, the **malloc** function returns a 0 (zero) if all memory map entries are already allocated; the driver should be coded to return EAGAIN in this case.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/malloc.c

SEE ALSO

mapinit(D3X), **mfree**(D3X), **sptalloc**(D3X), **sptfree**(D3X)

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 means of a space management memory map.

The example that follows shows how to allocate space from a private map. A fully-semaphored driver must initialize two semaphores: one for exclusive use of the map (`mapsema`, initialized to 1 in line 12) and one for blocking (`mapsemb`, initialized to 0 in line 13); these lines are not coded in a non-semaphored driver. Otherwise, the code for fully-semaphored drivers and non-semaphored drivers is the same:

- ❑ The driver allocates a buffer from the map (line 15).
- ❑ If the space allocation cannot be satisfied, the driver sets **u.u_error** to EAGAIN and returns (lines 16 and 17).
- ❑ The data is copied from the user data area to the allocated buffer (line 19).
- ❑ If an invalid address is detected in the user data are, the allocated buffer is released (line 20), and an error code is returned (lines 21 and 22).

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

struct map xx_map[XX_MAPSIZE];             /* Private buffer space map */
char xx_buffer[XX_BUFSIZE];                /* driver xx_ buffer area */

  :

register caddr_t addr;
register int size;
   size = min(u.u_count, XX_MAXSIZE);      /* Break large I/O request */
                                           /* into small ones */
   initsema(&mapsema, 1, 0);
   initsema(&mapsemb, 0, 0);
   mapinit(xx_map, sz, &mapsema, &mapsemb)

   if((addr = caddr_t)malloc(xx_map, size, 0)) == NULL) {
        u.u_error = EAGAIN;
        return;
   }                                       /* endif */

   if copyin(u.u_base, addr, size) == -1) {
        mfree(xx_map, size, addr);
        u.u_error = EFAULT;
        return;
   }                                       /* endif */
```

NAME

mapinit – initialize a private space management map

SYNOPSIS

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

mapinit(map, mapsize, s1, s2)
struct map *mp;
int mapsize;
int s1, s2;
```

ARGUMENTS

mp	memory map from where the resource is drawn
mapsize	number of entries for the memory map table
s1	semaphore to control map; set to 0 if no semaphoring is required
s2	synchronization semaphore (also called **mapout(map)**); set to 0 if no semaphoring is required

DESCRIPTION

The driver must initialize the map structure by calling the **mapinit** macro. 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 an object can actually be allocated from the memory map.

Through the **mapinit** macro, drivers may define private space management map for allocation of memory space and initialize a suspend lock semaphore to protect the map when it is accessed. 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.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None

LEVEL

Base or Interrupt

SOURCE FILE

sys/map.h

SEE ALSO

malloc(D3X), **mfree**(D3X), **sptalloc**(D3X), **sptfree**(D3X)

EXAMPLES

Fully Semaphored Driver

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 9). The private buffer area that is managed by the space management memory map is assigned to the memory map (line 10).

```
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   initsema(&mapsema, 1, 0);        /* Locking semaphore for map */
7   initsema(&mapout, 0, 0);         /* Synchronization semaphore */

8   /* Initialize space management map with number of slots in the map */
9   mapinit(xx_map, XX_MAPSIZE, &mapsema, &mapout);
10  mfree(xx_map, XX_BUFSIZE, xx_buffer); /* Initialize map */
11  /* with total buffer area it is to manage */
```

Non–Semaphored Driver

mapinit can also be used in non-semaphored drivers. In this case, the *s1* and *s2* parameters are both specified as 0. Note that it is not necessary to use synchronization functions to avoid contention because the operating system ensures that only one instance of the driver executes at a time.

```
#define XX_MAPSIZE  12
#define XX_BUFSIZE  2560

struct map xx_map[XX_MAPSIZE]; /* Private buffer for space map */
   char xx_buffer[XX_BUFSIZE]; /* Driver xx_buffer area */
      ⋮
mapinit(xx_map, XX_MAPSIZE, 0, 0); /* Initialize space management map */
                                   /* with number of slots in the map */
mfree(xx_map, XX_BUFSIZE, xx_buffer);               /* Initialize map */
                            /* with total buffer area it is to manage */
```

max(D3X)

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 macro returns the larger of two integers.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

The larger of the two numbers.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sysmacros.h

SEE ALSO

min(D3X)

EXAMPLE

```
extern int tthiwat[]; /* High water marks for cblock allocation base */
                       /* Baud rate (t_cflag & CBAUD) /*
extern struct tty xx_tty[];

  ⋮

register struct tty *tp = xx_tty[minor(dev)];
register int maxsize;

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

NAME

mfree – free space back into a private space management map

SYNOPSIS

#include<sys/map.h>

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

ARGUMENTS

mp	map pointer
size	number of units being freed
a	address of the buffer as allocated by **malloc**(D3X), given as an unsigned integer

DESCRIPTION

This function releases space back into a private space management map. It is the opposite of **malloc**, 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 reuse.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE None.

CAUTION

It is possible the map area will have insufficient space to record details of the freed buffer. In this case, the memory is lost to the system and the following warning message is displayed on the console:

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

where mp *is the hexadecimal address of the map structure;* size *is the number of buffers freed (in decimal); and* index *is the decimal address to the first buffer unit freed.*

This loss of memory occurs only under extraordinary conditions, which are not likely to be present in normal use. For example, if the driver allocated several hundred buffers by means of ***malloc****, then freed alternate buffers by means of* ***mfree****, the resultant fragmentation of the map would lead to loss of buffers as described here.*

LEVEL Base Only (Do not call from an interrupt routine)

SOURCE FILE *os/malloc.c*

SEE ALSO **malloc**(D3X), **mapinit**(D3X)

EXAMPLE For examples of using **mfree** in a fully-semaphored or a non-semaphored driver, refer to **malloc**(D3X).

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 macro returns the lesser of two integers.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

The lesser of the two numbers.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sysmacros.h

SEE ALSO

max(D3X)

EXAMPLE

The following example illustrates a use of **min**, to get the smaller buffer size.

```
size = min(u.u_count, cfreelist.c_size);
```

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;
```

ARGUMENTS

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.)

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

The internal minor number.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sysmacros.h

SEE ALSO

major(D3X), **makedev**(D3X)

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

```
struct device                    /* Physical device registers layout */
{
       int   control;            /* Physical device control word */
       int   status;             /* Physical device status word */
       short recv_char;          /* Receive character from device */
       short xmit_char;          /* Transmit character to device */
};                               /* end device */

extern struct device xx_addr[]; /* Physical device registers location */
extern int         xx_cnt;       /* Number of physical devices */
extern struct tty xx_tty[];

  :

register struct tty *tp = xx_tty[minor(dev)]; /* Get device's tty struct*/
register struct device *rp;

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

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

NAME

nodev -- indicate a driver routine is missing

SYNOPSIS

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

ARGUMENTS

None.

DESCRIPTION

This function is an internal function that marks the point(s) in the cdevsw(D4X) or bdevsw(D4X) switch table where a driver's primary routine was omitted. **nodev** should not be used by the driver developer; its description is provided here for informational purposes only.

SEMAPHORE RAMIFICATIONS

None.

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

NAME

NOT_ALIGNED – prevent compiler from reporting unaligned structures in the kernel

SYNOPSIS

```
NOT_ALIGNED
structure_definition {
        structure_members
}
```

ARGUMENTS

None.

DESCRIPTION

For processors on which alignment rules are not defined, the NOT_ALIGNED macro is used to prevent the compiler from reporting that structures in the kernel are not aligned on a word boundary. NOT_ALIGNED is used only when the kernel is being built. It is most commonly used when defining structures that give the physical layout of device registers, but is also sometimes used with definitions of software structures as well. For processors on which alignment rules are defined, this macro performs no action.[1]

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

Not applicable.

LEVEL

Not applicable.

SOURCE FILE

sys/types.h

[1] To determine if alignment rules are defined on your machine, refer to the Release Notes shipped with your system.

NAME

nulldev – perform no operation

SYNOPSIS

```
nulldev()
{
}
```

ARGUMENTS

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*).

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Not called from a driver.

SOURCE FILE

os/subr.c

NAME

olongjmp – return to location specified by **osetjmp**(D3X)

SYNOPSIS

```
olongjmp(save_area);
c_addr save_area;
```

ARGUMENTS

save_area area to which **osetjmp** saved the registers. This can never be **u.u_qsav**.

DESCRIPTION

The **olongjmp** function resets the registers saved by **osetjmp** from values in *save_area* and returns to the location from which **osetjmp** was called. It is seldom used in either drivers or system calls; usually the **klongjmp**(D3X) function is used when kernel code must return to a sane point.

SEMAPHORE RAMIFICATIONS

No semaphores should be held when calling **olongjmp**.

RETURN VALUE

If successful, **olongjmp** returns a value of 1.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

ml//cswitch.s*

SEE ALSO

klongjmp(D3X), **olongjmp**(D3X), **osetjmp**(D3X)

NAME

osetjmp – save registers and return location for **olongjmp**(D3X)

SYNOPSIS

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

osetjmp (save_area);
```

ARGUMENTS

save_area the area where registers and return location are to be saved. This argument cannot be **u.u_qsav**.

DESCRIPTION

The **osetjmp** function saves registers and a return location to which the **olongjmp** function will return control if called. It differs from **ksetjmp**(D3X) in that `u.u_qsav` is not used (the user passes the save area). It is rarely used.

SEMAPHORE RAMIFICATIONS

No semaphores should be held when **osetjmp** is called.

RETURN VALUE

If successful, **osetjmp** returns 0.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

ml//cswitch.s*

SEE ALSO

klongjmp(D3X), **olongjmp**(D3X), **osetjmp**(D3X)

NAME

386

outb, outw, outl − write data to a specified 80x86 I/O address (port)

SYNOPSIS

```
outb(port,value)
int port;
int value;
```

The synopses of **outw** and **outl** are the same as the synopsis of **outb**.

ARGUMENTS

port	address of the I/O space where the data is to be written
value	value to be written

DESCRIPTION

The function **outb**, **outw**, or **outl** writes a byte, a short (16-bit) value, or a long (32-bit) value, respectively, to the 80x86 I/O address space.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

sys/inline.h

SEE ALSO

in(D3X)

NAME

passc – pass character to user-level process

SYNOPSIS

```
passc(c)
char c;
```

ARGUMENTS

c	character to be passed

DESCRIPTION

passc passes a character back to the location pointed to by the `u.u_base` member of the user(D4X) structure and updates the `u.u_base`, `u.u_count`, and `u.u_offset` members of the user structure.

SEMAPHORE RAMIFICATIONS

No spin locks and no global semaphores should be held when calling **passc**.

RETURN VALUE

passc returns the updated value of `u.u_count`. On the last character of the user's read operation, **passc** returns −1. If **passc** cannot write to the address specified by **u.u_base**, it returns −1 and sets **u.u_error** to EFAULT.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/move.c

SEE ALSO

cpass(D3X), user(D4X)

NAME

pg_getaddr – get page address

SYNOPSIS

```
unsigned int
pg_getaddr(pde)
pde_t *pde;
```

ARGUMENTS

pde the address of a page descriptor entry

DESCRIPTION

This macro extracts the physical address of the page mapped by the page descriptor, *pde*.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

The physical address mapped by the specified page descriptor.

LEVEL

Base or Interrupt

SOURCE FILE

sys//immu.h*

NAME

physck – verify the requested block exists

SYNOPSIS

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

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

ARGUMENTS

nblocks number of logical blocks in the partition

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

The following members in the user structure are implicit arguments to **physck**:

u.u_offset	a byte offset in the file
u.u_count	a byte count for the transfer
u.u_ap	points to the original parameters of the system call.

These members are used the same as with standard read and write calls (that is, a file descriptor, a buffer address, and a count).

DESCRIPTION

physck is used in the block driver **read**(D2X) and **write**(D2X) routines to verify that 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. 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.

The *nblocks* parameter is used by **physck** to calculate the number of bytes held in the partition. If the desired offset is past the end of the partition, then ENXIO is set in **u.u_error** and a 0 is returned.

If the desired offset is exactly at the end of the partition, the *rwflag* is checked:

- ❑ If the flag indicates a write operation, then ENXIO is set and 0 is returned.

- □ If the flag indicates a read, 0 is returned (no error code is set in **u.u_error**). If the caller proceeds no further, this will result in correct end-of-file handling.

If the required transfer length would take the transfer past the end of the partition, then **physck** alters various fields to ensure that the transfer remains within bounds. It adjusts **u.u_count** and also the byte count parameter to the original system call, reducing them so that the transfer goes exactly to the limits of the partition.

NOTE

***physck** is appropriate only in response to a genuine **read**(2) or **write**(2) system call. It is inappropriate to use **physck** in other circumstances, such as to implement custom I/O controls.*

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **physck**.

RETURN VALUE

A return of 1 indicates that a transfer may go ahead. The transfer may not be exactly as originally requested; if it would go beyond the limits of the partition, then the transfer count in **u.u_count** is reduced, as is the *count* parameter to the original system call, as described above.

A return of 0 indicates that no transfer is possible. This may be due to a read at end-of-file, in which case no error is reported. Otherwise, **u.u_error** is set to ENXIO.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/physio.c

SEE ALSO

KPG, "Synchronized I/O Operations"
dma_breakup(D3X), **physio**(D3X)

EXAMPLE

For an example of the use of **physck**, refer to the example given for **dma_breakup**(D3X).

NAME

physio – call **strategy**(D2X) routine to process raw I/O for block interface drivers

SYNOPSIS

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

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

ARGUMENTS

strat conceptually, the address of the driver's **strategy**(D2X) routine, which **physio** uses to determine appropriate parameters. The more typical usage is for the caller to supply the address of a subroutine or function that performs some other device-dependent operations (such as calling **dma_breakup**(D3X)) before calling the driver's **strategy** routine.

bp address of a buf(D4X) header. It is not necessary to supply a buf header, and the typical usage of **physio** is with this parameter set to 0. If a buf header is supplied, it is used in passing the data to the supplied **strategy** routine, with various fields updated as required. If no buf header is supplied, **physio** obtains one, freeing it after the I/O operation is complete.

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.

Also note that the following members from the user(D4X) structure are implicit arguments to **physio**:

u.u_base	transfer buffer start address
u.u_count	transfer count
u.u_offset	position in file
u.u_procp	pointer to proc(D4X) structure

DESCRIPTION

The **physio** function locks the area of user virtual memory so that transfers may take place directly between the device and user memory without worrying about paging (refer to **userdma**(D3X) for a function that performs this directly). If an error occurs in the locking of memory, then **physio** returns immediately with an error (EFAULT) set in **u.u_error**.

Once the user virtual memory is locked, **physio** sets up a `buf`(D4X) header describing the operation. The members in `buf` are set as follows:

b_flags	set to `B_BUSY | B_PHYS | rwflag`
b_error	cleared to zero
b_proc	set from **u.u_procp**
b_dev	set from the parameter *dev*
b_un.b_addr	set from **u.u_base**
b_blkno	set indirectly from **u.u_offset** (converted from bytes to logical disk blocks)
b_bcount	set from **u.u_count**

The contents of all other fields in the `buf` are undefined.

The **physio** function then calls the supplied *strat* routine, passing as the single parameter a pointer to the `buf`(D4X) header. It then blocks on the **b_iodone** semaphore. For normal transfers, when the transfer is complete, **physio** is unblocked by the driver interrupt routine through the **iodone**(D3X) function. If the driver detects any errors that prevent it from starting the I/O transfer, it must call **iodone**(D3X) to unblock the **physio** function.

After being unblocked, **physio** unlocks the user virtual memory. It then checks the contents of the `buf` header. The **u.u_count** field is updated with the contents of the `buf` **b_resid** field. In addition, if an error is reported *via* the B_ERROR flag, the **u.u_error** field is updated from the **b_error** field of the `buf`.

If a `buf` was supplied, then the only clean up performed by **physio** is to ensure that the B_BUSY and B_PHYS flags are not set. All other fields are as left by the **strategy** routine. If a buffer was not supplied and **physio** had to supply a temporary buffer, then it is replaced in a free buffer pool.

As a note to driver writers, the data address given by **physio** is typically a user virtual memory address. This can be determined by looking at the **u.u_segflg** field of the `user` area.

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.

Conventionally, in the absence of performance constraints, intermediate kernel buffering is used as a method of avoiding the complication of dealing with the possibly discontiguous user memory. The **dma_breakup**(D3X) function can be used for this work. Alternatively, the **disjointio**(D3X) function can be used to obtain the real addresses of the pages that make up the user's buffer area.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **physio**.

RETURN VALUE

physio does not have an explicit return value, but may update **u.u_error** with an appropriate error code, and **u.u_count** with the number of bytes not transferred from **b_resid**.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/physio.c

SEE ALSO

KPG, "Synchronized I/O Operations"
dma_breakup(D3X), **physck**(D3X), **strategy**(D2X)

EXAMPLE

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

NAME

pnum – get page number

SYNOPSIS

```
pnum(addr)
unsigned int addr;
```

ARGUMENTS

addr address for which the page number is to be returned

DESCRIPTION

pnum returns the page number of the specified address. This value is the virtual address divided by the page size.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

page number

LEVEL

Base or Interrupt

SOURCE FILE

sys//immu.h*

SEE ALSO

poff(D3X), **psnum**(D3X), **snum**(D3X), **soff**(D3X)

poff(D3X)

NAME poff – get page offset

SYNOPSIS

```
poff(addr)
unsigned int addr;
```

ARGUMENTS *addr* address for which the offset is to be returned

DESCRIPTION **poff** returns the page offset of the specified address.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE page offset

LEVEL Base or Interrupt

SOURCE FILE *sys/*/immu.h*

SEE ALSO **pnum**(D3X), **psnum**(D3X), **snum**(D3X), **soff**(D3X)

NAME

popsr – enable interrupts and restore saved interrupt privilege level (ipl)

SYNOPSIS

```
popsr()
```

DESCRIPTION

popsr reenables all interrupts enabled before **pushsrdisable**(D3X) is called. **popsr** also restores the interrupt privilege level (ipl) saved by **pushsrdisable**.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

sys/inline.h

SEE ALSO

enable(D3X), **pushsrdisable**(D3X), **spsema**(D3X), **svsema**(D3X)

NAME

preiowait – suspend execution pending completion of a block or raw I/O request for a block access device

SYNOPSIS

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

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

ARGUMENTS

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

DESCRIPTION

The **preiowait** function is typically used to block in the **strategy**(D2X) routine when processing is required that can be performed only after the operation is complete. For example, it is used to block in **dma_breakup** to allow data to be copied and buffers freed.

Under UNIX System V, an **iowait**(D3X) system call is issued to wait for an I/O operation that uses a buffer header. On a non-semaphored kernel, the process sleeps until the B_DONE flag in **b_flags** is set; **preiowait** could be called multiple times during a single operation. The first call waits until the driver calls **iodone**(D3X), and subsequent **iowait** calls just return when they find the B_DONE bit already set.

The buffer header structure on the REAL/IX Operating System includes a semaphore, **b_iodone**. To wait for the I/O operation to complete, **iowait** does a **psema**(D3X) on the `bp->b_iodone` and blocks until **iodone** issues the corresponding **vsema**(D3X) indicating that the operation is complete. Multiple **iowait** calls cannot be performed because each one performs a **psema** operation to decrement the value of `bp->b_iodone`, but the **iodone** function issues only one **vsema** call to increment the value of `bp->b_iodone`. The first additional **iowait** call would block "forever" because no additional **iodone** calls are forthcoming.

preiowait issues a **psema** call to wait for the operation to complete, then issues a **vsema** on `bp->b_iodone` to prevent the next **iowait** call from hanging. If multiple **iowait** calls are needed in a code sequence for a buffer header, all but the last one must be **preiowait**, and the last one must be **iowait**.

For raw access, **physio**(D3X) issues the final **iowait** call; for block access, the **iowait** call is performed by the higher-level routines after the driver **strategy**(D2X) routine returns.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **preiowait**.

RETURN VALUE

None. The buffer header's **b_iodone** semaphore is left with a value of −1. Before the buffer is released, an **iowait**(D3X) call must be issued to increment the value of the semaphore to 0.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/bio.c

SEE ALSO

delay(D3X), **iodone**(D3X), **iowait**(D3X), **psema**(D3X), **timeout/timeoutfs**(D3X), **ttywait**(D3X), **untimeout**(D3X), **vsema**(D3X)

NAME

psema, rpsema, ppsema – lock semaphore for a resource

SYNOPSIS

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

val = psema(sem_addr, flags);
sema_t *sem_addr;
int flags;
```

The synopses for **rpsema** and **ppsema** are identical to the synopsis for **psema**.

ARGUMENTS

sem_addr identifies the semaphore to be locked

flags determine how the process that called **psema** reacts to interrupt signals and if the priority boost is to be applied; valid *flags* values are:

0 Wait may not be interrupted by signals and boosting algorithm should not be used.

SEMINTR Check for signals before suspending self and after being resumed. If no signals are held or ignored and if SEMCATCH is not specified, **klongjmp** will be invoked (This is roughly equivalent to a **sleep** priority greater than PZERO).

SEMCATCH Check for signals before suspending self and after being resumed. If there are signals, return error code (1 or −1); otherwise, return 0. SEMCATCH implies SEMINTR.

SEMRTBOOST Apply a boosting algorithm that temporarily boosts the priority of lower priority process when it holds the semaphore if the semaphore is needed by a higher priority realtime process. This flag should be applied only to semaphores that are expected to be used by realtime processes after their initialization time processing.

No other flags can be used with SEMRTBOOST, and **vsema**(D3X) calls for this semaphore must also include the SEMRTBOOST flag.

SEMINTBOOST Perform interactive boost (boosting for non-realtime processes). SEMINTBOOST should be used only for terminals (tty). SEMINTBOOST implies SEMINTR.

SEMNOLOOP If an interrupt signal that is held or ignored has made the process runnable, return a value of 1. Without this flag, if **psema** determines that the process was interrupted by a non-ignored or held signal, it causes the process to block again. SEMNOLOOP implies SEMINTR. It is commonly used with interruptible blocks that use a counter to ensure an appropriate value for the semaphore.

DESCRIPTION

The **psema** family of macros decrements the value of the semaphore specified by *sem_addr*. If the value of the semaphore becomes negative, the executing process is suspended and placed on a linked list of processes sleeping on the semaphore.

If interrupt signals are pending against a blocked process, the value of the *flags* parameter determines whether they are deferred or caught.

- If *flag* is SEMINTR, receipt of a signal will cause a **klongjmp**(D3X) operation. Without this flag, the blocked process will not be awakened by an interrupt signal. SEMINTR is implied by SEMCATCH, SEMINTBOOST, and SEMNOLOOP.

- If *flag* is SEMCATCH, the signal is caught and handled according to code written in the driver. **psema** returns a value that indicates whether or not the operation was successful.

For guidelines on selecting the correct *flags*, refer to the *Kernel Programming Guide*.

If ***psema*** *is called from the driver* ***strategy****(D2X) routine, use the SEMCATCH flag.*

⚠ CAUTION

Semaphores that are blocked with the SEMINTR or SEMCATCH flag may need to be reinitialized with ***reinitsema****(D3X) before the first* ***psema*** *call that is expected to block because the value of the semaphore will be incremented by all interrupts received as well as by the* ***vsema*** *function. The driver must maintain a count of processes blocked because the semaphore cannot be reinitialized if a process is already blocked.*

Semaphores decremented with **psema** can be incremented with the **vsema**(D3X) macro. If the **psema** call uses no flags (0), the semaphore can also be incremented with **cvsema**(D3X).

The **rpsema** and **ppsema** macros are faster than **psema** and can be used to optimize performance in the driver. **rpsema** can be used if interrupts are already disabled with the **splhi** function. **ppsema** can be used if all interrupts are enabled.

SEMAPHORE RAMIFICATIONS

Drivers that call **psema** must be installed fully semaphored. No spin locks should be held when calling **psema**.

RETURN VALUE

The **psema** functions return a value only if the SEMINTR flag (or a flag that implies SEMINTR) is specified. Return values are:

0 operation was successfully performed; the process has the resource.

−1 operation was not performed because a there is a non-ignored, non-held signal pending for the process.

1 operation was not performed, but a non-ignored, non-held signal is not pending for the process (is returned only if the SEMNOLOOP flag is specified as well as SEMCATCH).

For other flags, the return value is undefined.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

sys/sema.h

SEE ALSO

KPG, "Synchronization"
cpsema(D3X), **cvsema**(D3X), **decsema**(D3X), **incsema**(D3X), **klongjmp**(D3X), **valusema**(D3X), **vsema**(D3X)

NAME

psignal – send signal to a process

SYNOPSIS

```
#include <sys/signal.h>
#include <sys/immu.h>
#include <sys/sema.h>
#include <sys/region.h>
#include <sys/psw.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* should be in the range of 1 to (NSIG−1). 0 and numbers greater than or equal to NSIG are also valid values, indicating that no signal is to be sent. NSIG and valid signals are listed in *signal.h*.

DESCRIPTION

This function is called by the driver 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 is blocked by a **psema**(D3X) with the SEMINTR flag[1], **psignal** makes the process executable. Once the process executes, a **klongjmp**(D3X) is executed, which returns to **u.u_qsav**.

If the driver needs to do cleanup before the **klongjmp**, it should block with the SEMCATCH flag, which implies SEMINTR. In this case, the driver does any necessary cleanup, then issues the **klongjmp** call.

psignal is retained here for compatibility; **psignalcur** and **psignalval** are faster ways to provide the same functionality.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **psignal**.

[1] If the driver is installed under CPU affinity, major-device semaphoring, or minor-device semaphoring, **psignal** sends the signal unless the process has called **sleep**(D3X) to wait at a priority higher than PZERO. If PZERO has not been ORed with PCATCH, **psignal** issues **klongjmp**. If PZERO has been ORed with PCATCH, the driver does any necessary cleanup, then calls **klongjmp**. PZERO is defined in *param.h* and **p_pri** is explained on the proc(D4X) manual page.

RETURN VALUE None.

LEVEL Base or Interrupt

SOURCE FILE *os/sig.c*

SEE ALSO **psignalcur**(D3X), **psignalval**(D3X), **send_event**(D3X), **signal**(D3X)

EXAMPLE In the following example:

- ❑ Get device registers (line 12) and get port number (line 13).
- ❑ 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).
- ❑ Note that a more efficient way of providing the same functionality is to use **psignalcur**(D3X).

```
 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];
13  register int  port = minor(dev) & 0x07;

14    ⋮

15     if ((rp->modem_status & (0x0100 << port)) == 0)
16     {
17       psignal(u.u_procp, SIGHUP);
18       return;
19     }                              /* endif */
```

NAME

psignalcur – send a valid signal number to the currently executing process

SYNOPSIS

```
#include <sys/signal.h>
#include <sys/immu.h>
#include <sys/sema.h>
#include <sys/region.h>
#include <sys/psw.h>

psignalcur(p, signum)
struct proc *p;
int signum;
```

ARGUMENTS

p pointer to the proc(D4X) structure of the process being signaled, in other words, **u.u_procp**

signum signal macro name that expands to an integer constant expression. Refer to **sigset**(2) for a list of valid signals; valid signal numbers are listed in *signal.h*.

DESCRIPTION

psignalcur sends a valid signal number to the currently executing process. It is significantly faster than **psignal**(D3X).

If the driver needs to do cleanup before the **klongjmp**, it should block with the SEMCATCH flag, which implies SEMINTR. In this case, the driver does any necessary cleanup, then issues the **klongjmp** call.

SEMAPHORE RAMIFICATIONS

No spin locks should be set when calling **psignalcur**.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

sys/proc.h

SEE ALSO

psignal(D3X), **psignalval**(D3X), **send_event**(D3X), **signal**(D3X)

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];
13  register int  port = minor(dev) & 0x07;

14    ⋮

15   if ((rp->modem_status & (0x0100 << port)) == 0)
16   {
17     psignalcur(u.u_procp, sigbit(SIGHUP));  */
18     return;
19     }
```

NAME

psignalval – send a valid signal number to any process

SYNOPSIS

```
#include <sys/signal.h>
#include <sys/immu.h>
#include <sys/sema.h>
#include <sys/region.h>
#include <sys/psw.h>

psignalval(p, signum sigmask)
struct proc *p;
int signum, sigmask;
```

ARGUMENTS

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

signum signal macro name that expands to an integer constant expression; refer to **sigset**(2) for a list of valid signals.

sigmask mask of signal sent, defined as `sigtomask(signum)`. The definition of **sigtomask** is:

```
#define sigtomask(n)    (1L<<(n-1))
```

Valid signal names and numbers are listed in *signal.h*.

DESCRIPTION

psignalval sends a valid signal number to any process. **psignalval** is faster than **psignal**(D3X), but not as fast as **psignalcur**(D3X). If the process being sent the signal is blocked by a **psema**(D3X) with the SEMINTR flag[1], **psignalval** makes the process executable by executing **klongjmp**(D3X), which returns to **u.u_qsav**.

If the driver needs to do cleanup before the **klongjmp**, it should block with the SEMCATCH flag, which implies SEMINTR. In this case, the driver does any necessary cleanup, then issues the **klongjmp** call.

SEMAPHORE RAMIFICATIONS

The **p_lock** member of the proc(D4X) structure must be locked by the caller before **psignalval** is called.

[1] If the driver is installed under CPU affinity, major-device semaphoring, or minor-device semaphoring, **psignalcur** sends the signal unless the process has called **sleep**(D3X) to wait at a priority higher than PZERO. If PZERO has not been ORed with PCATCH, **psignalcur** issues **klongjmp**. If PZERO has been ORed with PCATCH, the driver does any necessary cleanup, then calls **klongjmp**. PZERO is defined in *param.h* and **p_pri** is explained on the proc(D4X) manual page.

RETURN VALUE None.

LEVEL Base or Interrupt

SOURCE FILE *sys/proc.h*. **sigtomask** is defined in *sys/signal.h*.

SEE ALSO **psignal**(D3X), **psignalcur**(D3X), **send_event**(D3X), **signal**(D3X), **sigset**(2)

psnum(D3X)

NAME

psnum – get page number within the segment

SYNOPSIS

```
psnum(addr)
unsigned int addr;
```

ARGUMENTS

addr address for which the page number within the segment is to be returned

DESCRIPTION

psnum returns the page number within the segment for the specified address.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

page number within the segment

LEVEL

Base or Interrupt

SOURCE FILE

sys//immu.h*

SEE ALSO

pnum(D3X), **poff**(D3X), **snum**(D3X), **soff**(D3X)

NAME

pushsrdisable – disable interrupts and save current ipl

SYNOPSIS

```
pushsrdisable()
```

DESCRIPTION

pushsrdisable disables all interrupts for the CPU on which code is executing and saves the current interrupt privilege level (ipl) to be restored with **popsr**(D3X). **pushsrdisable** is useful for protecting a local resource with less overhead than the other functions entail. **pushsrdisable** also allows the current ipl to be restored when interrupts are reenabled.

NOTE *Disabling interrupts for long periods of time will degrade general system performance.*

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

sys/inline.h

SEE ALSO

disable(D3X), **popsr**(D3X), **spsema**(D3X), **svsema**(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 none are allocated for the clist or because the last clist is full, **putc** retrieves a new cblock from the cfreelist(D4X).

SEMAPHORE RAMIFICATIONS

Drivers calling **putc** must be installed under the compatibility modes.

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, the system panics. (Note that the number of cblocks in the system can be specified with the tunable parameter NCLIST.)

LEVEL

Base or Interrupt

SOURCE FILE

io/clist.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
clist(D4X), **getc**(D3X), **getcb**(D3X), **getcf**(D3X), **putcb**(D3X), **putcf**(D3X)

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 6).
- If the user area contains an invalid address, **fubyte** returns an error code (line 7).
- Otherwise, add the byte to the last cblock in the clist (line 10) and update number of bytes remaining (line 11).

```
 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) {
 6           if ((c = fubyte(u.u_base++)) == -1) {
 7                   u.u_error = EFAULT;
 8                   return;
 9           }
10            putc(c, &tp->t_outq);
11            u.u_count--;
12   }
```

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 links the cblock specified by *cbp* to the clist specified by *clp* and increases the character count in the clist head by the number of the characters in the cblock.

SEMAPHORE RAMIFICATIONS

Drivers calling **putc** must be installed under the compatibility modes.

RETURN VALUE

putcb always returns a 0 (zero).

LEVEL

Base or Interrupt

SOURCE FILE

io/clist.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
cblock (D4X), clist(D4X), **getc**(D3X), **getcb**(D3X), **getcf**(D3X), **putc**(D3X), **putcf**(D3X)

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).

❑ Copy the data from the user data area to the allocated cblock (line 11).

❑ If an invalid address is detected in the user data area, return the cblock to the cfreelist (line 13) and return an error code.

- Otherwise, change the input index **c_last** to the number of the characters in cblock (line 17).

- Change the output index **c_first** to show that no characters have been removed from the cblock (line 18).

- Add the cblock to the end of the clist (line 19).

- The pointer to the user data area is advanced to the next starting byte of data to be copied (line 20), and the remaining byte count is updated (line 21).

```
 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)
 7  {
 8     size = min(u.u_count, cfreelist.c_size);  /* Get smaller buffer size */
 9
10     cp = getcf()                        /* Get free cblock from freelist */

11     if (copyin(u.u_base, cp->c_data, size) == -1)
12     {
13       putcf(cp);
14       u.u_error = EFAULT;
15       return;
16     }
17     cp->c_last = size;

18     cp->c_first = 0;

19     putcb(cp, tp->t_outq);
20     u.u_base += size;
21     u.u_count -= size;
22  }
```

NAME putcf – put cblock(D4X) on the free list

SYNOPSIS

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

ARGUMENTS *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).

SEMAPHORE RAMIFICATIONS

Drivers calling **putcf** must be installed under the compatibility modes.

RETURN VALUE None.

LEVEL Base or Interrupt

SOURCE FILE *io/clist.c*

SEE ALSO *KPG*, "Drivers in the TTY Subsystem"
cblock(D4X), **getc**(D3X), **getcb**(D3X), **getcf**(D3X), **putcb**(D3X), **putcf**(D3X)

EXAMPLE Refer to the example given for **getcb**(D3X).

NAME

rel_timer – release interval timer

SYNOPSIS

```
int rel_timer(tp);
struct tmr *tp;
```

ARGUMENTS

tp pointer to `tmr` structure to be released

DESCRIPTION

The **rel_timer** function releases the interval timer obtained with **get_timer**(D3X) and returns it to the pool of available interval timers. The resource is then available for use by another driver. If *tp* does not point to an allocated interval timer, **rel_timer** returns EINVAL; otherwise, it returns a zero.

CAUTION

rel_timer *performs minimal parameter checking. Calling* **rel_timer** *with a bad value for* tp *or releasing the same timer more than once will have undefined – and probably fatal – consequences.*

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

If successful, **rel_timer** returns 0; otherwise, if *tp* is not an allocated timer, **rel_timer** returns EINVAL.

LEVEL

Base or Interrupt

SOURCE FILE

os/timer.c

SEE ALSO

get_timer(D3X), **set_timer**(D3X)

NAME

rtuser – verify realtime permission mode

SYNOPSIS

```
rtuser();
```

ARGUMENTS

None.

DESCRIPTION

This function determines if the current user has realtime permissions.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

If the current user has realtime permissions, 1 is returned. Otherwise, 0 (zero) is returned and the driver should set **u.u_error** to EPERM (not owner).

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

sys/user.h

SEE ALSO

suser(D3X), **useracc**(D3X)

EXAMPLE

Using **rtuser** is straightforward, easy, and viable for many situations. The following example shows such a test. Note that because the superuser permissions are adequate to do anything that requires realtime permissions, the test for realtime permissions should be used in conjunction with **suser**(D3X); the example shows a typical idiom of programs written to run under the REAL/IX Operating System.

If **suser**(D3X) fails, **u.u_error** is set to EPERM by the operating system, so the driver does not need to set this error.

```
if (!(rtuser() || suser())){
      return;
}
```

NAME

selwakeup – unblock processes waiting to select a device

SYNOPSIS

```
selwakeup(proc, coll)
```

ARGUMENTS

proc address of process to be unblocked

coll collision flag; if set, more than one process simultaneously attempted to select this device and needs to be awakened.

DESCRIPTION

selwakeup is used in drivers that have a **select**(D2X) entry point to select the **select**(2) system call. **selwakeup** is usually called from the driver's **intr**(D2X) routine[1] when a device becomes accessible for the access required (read or write) and status in the driver-specific select structure (described on the **select**(D2X) manual page) indicates that one or more processes are waiting for the device to become accessible for this type of access.

Processes that have attempted to select a device controlled by the driver and found the device not selectable will update the data structures with the appropriate information. The process may block sometime after calling the driver's **select**(D2X) routine because none of the devices it tried to select were selectable. **selwakeup** unblocks those processes. If the process is not blocked, **selwakeup** just returns.

selwakeup is passed two arguments. The first argument is the address of the proc(D4X) structure for the process that is trying to select the device. This is the information in the "read-select" or "write-select" members of the driver-specific select data structure, depending on whether the device became readable, writable, or both.

NOTE *__selwakeup__ is called to unblock either a read select or a write select. If a device interrupt occurs and it is determined that the device has become both readable and writable and both conditions are being selected for, __selwakeup__ must be called twice.*

After calling **selwakeup**, the driver should clear the appropriate proc structure address field and collision flag within its data structures for the device to prevent more unnecessary **selwakeup** calls.

All accesses to the driver's select data structure must be protected to avoid race conditions while testing and modifying these fields because the same fields are also accessed by the driver's **select**(D2X) routine. Fully-

[1] **selwakeup** can be called from the base level of the kernel as well. This approach is used for drivers that use a daemon to process deferred interrupts.

semaphored drivers usually use a spin lock (**spsema**(D3X)), drivers installed under CPU affinity usually use an **spl**(D3X) call, and drivers installed under major- or minor-device semaphoring do not need to explicitly protect the structure. Note the following:

- □ The protection must begin prior to the modification of the "this device is readable/writable" fields (which are tested by the driver's **select** routine).
- □ The protection may be abandoned after the "selecting proc address" fields and the corresponding collision flags (which are modified by the driver's **select** routine) have been cleared.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base or Interrupt (Usually called from the interrupt handling routine)

SOURCE FILE

os/berk.c

SEE ALSO

select(D2X)

EXAMPLE

Refer to **select**(D2X) for an example of the **selwakeup** function.

NAME

send_event, SEND_EVENT – post event to user-level process

SYNOPSIS

for send_event:

```
#include <sys/proc.h>
#include <sys/errno.h>
#include <sys/immu.h>
#include <sys/region.h>
#include <sys/evt.h>

send_event(p, eid, type, ditem)
struct proc *p;
uint eid;
int type;
long ditem;
```

for SEND_EVENT:

```
#include <sys/proc.h>
#include <sys/errno.h>
#include <sys/immu.h>
#include <sys/region.h>
#include <sys/evt.h>
#include <sys/evtmacros.h>

SEND_EVENT(_p, _eid, _type, _ditem, _status)
struct proc *_p;
uint _eid;
int _type;
long _ditem;
int _status;
```

ARGUMENTS

p the process to which to post the event

eid the event identifier to post

type identifies the subsystem that sent the event. Valid values are:

EVT_TYPE_USER	user-posted event
EVT_TYPE_ASNCIO	asynchronous I/O completion event
EVT_TYPE_TIMER	timer expiration event
EVT_TYPE_INTR	connected interrupt occurred
EVT_TYPE_RES	resident process violation

ditem optional 32-bit data item to post with the event

_status the location where the return status will be stored

DESCRIPTION

send_event posts an event to the specified user-level process and event identifier. Before calling **send_event**, the driver must lock `p->p_lock`.

Note that kernel-level processes (including drivers) can post events to any user-level process on the system, not just processes associated with the driver. Caution should be exercised to ensure that no stray events are posted.

SEND_EVENT is an inline (macro) version defined in *sys/evtmacros.h*. It provides the same functionality as **send_event** but takes an additional argument, *_status*. The macro is useful when the calling process has at least two register pointers to spare. It is suggested that the process to which the event is to be posted also be stored in a register. The driver must be careful not to pass arguments that are evaluated twice (e.g., `++_ditem`). Before calling **SEND_EVENT**, the driver must lock `p->p_lock`.

SEMAPHORE RAMIFICATIONS

`p->p_lock` must be locked when calling **send_event**, and `slp_cnt_lock` and `rqlock` (defined in *sys/systm.h*) must not be locked.

RETURN VALUE

If successful, **send_event** returns 0. If unsuccessful, **send_event** will return one of the following error codes:

EAGAIN	process *p* is ignoring the signal
ENOSPC	process could not allocate space for the event block

SEND_EVENT returns the same values as **send_event** and additionally stores the return value at the location specified by the *_status* argument for reuse as a signal mask.

LEVEL

Base or Interrupt

SOURCE FILE

os/evt.c (**send_event**); *sys/evtmacros.h* (**SEND_EVENT**)

SEE ALSO

Programmer's Guide
evget(2), **evpost**(2), **evrcv**(2), **evrcvl**(2)
psignal(D3X), **psignalcur**(D3X), **psignalval**(D3X), **signal**(D3X)

EXAMPLE

The following code example is used to post a resident memory violation event:

```
evtdataitem |= DATUNLOCK;

if ((change > 0) && (eid != -1)) {
     register proc_t *p = u.u_procp;

     /* post event eid */
     pspsema(&p->p_lock);
     send_event(p, eid, EVT_TYPE_RES, evtdataitem);
     psvsema(&p->p_lock;
}
```

NAME

set_timer – set interval timer

SYNOPSIS

```
int set_timer(tp,val,func,funcarg);
struct tmr *tp;
struct itimerstruc *val;
void (*func) ();
char *funcarg;
```

ARGUMENTS

tp	pointer to the `tmr` structure allocated to this driver
val	pointer to the structure that holds the expire and repeat time for the timer
func	pointer to the function to be executed when the timer expires
funcarg	pointer to the argument to *func*

DESCRIPTION

The **set_timer** function sets the interval timer expiration value relative to the current time as specified in the structure pointed to by *val* and sets the timer running.

The expiration time and the repeat interval are stored and maintained in units of seconds and nanoseconds. If the expiration time in the structure pointed to by *val* is 0, the timer is disabled and removed from the active timer queue. It is not necessary to disable a timer before resetting its expiration value; the driver simply issues **set_timer** again with *val* pointing to the new expiration time.

If the call to **set_timer** is successful, it returns 0. Otherwise, if *tp* does not point to an allocated interval timer, **set_timer** returns EINVAL. It also returns EINVAL if either the delay or the repeat interval specified in the structure pointed to by *val* is greater than the maximum supported by the underlying timer type, or if either of the nanosecond fields of that structure contains an invalid value.

CAUTION

set_timer *performs minimal parameter checking. Calling* **set_timer** *with a* tp *parameter that was not obtained with* **get_timer** *or after the timer has been released by* **rel_timer** *will have undefined – and probably fatal – results.*

CAUTION

*When the timer expires, the user-supplied function (*func*) is called in the context of a kernel daemon. At some point, the daemon will be committed to calling this function. It is possible for a timer to be cancelled after the daemon is committed to calling the function but before the function completes execution. When writing a driver, you must be aware of the race conditions that result from this situation.*

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

If successful, **set_timer** returns 0. **set_timer** returns EINVAL under any of the following conditions:

- *tp* is not an allocated timer
- the delay value stored in *val* exceeds the maximum supported by the timer type
- the repeat interval value stored in *val* exceeds the maximum supported by the timer type
- *val* contains an invalid value in one of its nanosecond fields

LEVEL

Base or Interrupt; however, it is recommended that **set_timer** be used only in base-level code because of the CPU time it uses

SOURCE FILE

os/timer.c

SEE ALSO

get_timer(D3X), **rel_timer**(D3X)

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 the BREAK was received. 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*.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **signal**.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

os/sig.c

SEE ALSO

Programmer's Guide
psignal(D3X), **psignalcur**(D3X), **psignalval**(D3X), **send_event**(D3X), **sigset**(2)

EXAMPLE

In a terminal interrupt routine (**intr**(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 0s (zeros), this signifies a BREAK key was pressed (line 22).
- ❑ Therefore, send an interrupt signal to all processes in the process group (line 24).

```
struct device                         /* Physical device register location */
{
       int   control;                  /* Physical device control word */
       int   status;                   /* Physical device status word */
       short recv_char;                /* Receive character from device */
       short xmit_char;                /* Transmit character to device */
};

extern struct tty xx_tty[];            /* Logical device structure */
extern struct device xx_addr[];        /* Physical device registers */
extern int  xx_cnt;                    /* Physical device number */

  :

xx_intr(board)
int board;
{
register struct device *rp = xx_addr[board];  /* Get device register */
register struct tty *tp;
register int c, port;

while((c = rp->recv_char) & DATAVALID) != 0)
{
   port = (c >> 8) & 0x7;              /* Get terminal's port number */
   tp = &xx_tty[(board << 3) & port]; /* Get corresponding structure */
   if ((c & FRERROR) != 0 && (c & 0xff) == 0)
   {
     signal(tp->t_pgrp, SIGINT);
     ttyflush(tp, (FREAD | FWRITE));
     continue;
   }
}

  :
```

NAME

sleep – suspend process activity pending execution of a wakeup (not used in fully semaphored drivers)

SYNOPSIS

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

ARGUMENTS

addr 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 **klongjmp**(D3X) on receipt of a signal. Instead, it returns the value 1 to the calling routine.

DESCRIPTION

The **sleep** 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 calls **sleep** to wait for data to become available. 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 *addr* specified as that used to call **sleep**.

The *addr* 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 had 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 by doing a **klongjmp**(D3X) back to the system call entry code. It returns the value 1 to the calling routine.

To pick the correct **sleep** priority, decide 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 can use a priority numerically less than PZERO. (However, priorities less than or equal to PZERO should be used only if the 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 return. 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 **klongjump** 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 clean up any locked structures before calling **klongjmp**.

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.*

COMPATIBILITY

The **sleep** function is one of the traditional UNIX synchronization mechanisms; for compatibility with other UNIX-based operating systems, it is supported on computers that run under the REAL/IX Operating System. Drivers being ported to the REAL/IX Operating System from another system can use **sleep** if they are installed under one of the compatibility modes. Drivers that are not installed under a compatibility mode should not use

sleep but should use semaphore operations to block a process. The *Driver Development Guide* describes the compatibility modes and how to provide **sleep/wakeup** functionality with kernel semaphores.

Note that a driver that calls **sleep** should avoid calling any semaphoring functions and vice versa. Mixing synchronization methods in one driver may result in deadlocks.

SEMAPHORE RAMIFICATIONS

Drivers that call **sleep** must be installed under the compatibility modes.

RETURN VALUE

If the **sleep** *priority* argument is ORed with the defined constant PCATCH, the **sleep** function does not call **klongjmp** 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)

SOURCE FILE

os/slp.c

SEE ALSO

KPG, "Synchronization"
DDG, "Porting Drivers"
delay(D3X), **iodone**(D3X), **iowait**(D3X), **psema**(D3X), **timeout**(D3X), **ttywait**(D3X), **untimeout**(D3X), **wakeup**(D3X)

EXAMPLE

The following code is from a TTY driver that supports a dual console. It tests whether the port is currently being used as a dual console and, if it is, puts the process to sleep.

```
if (Dconcurrent) {
     while (xxxx_state[dev] & DCON) {
          sleep((caddr_t) & tp->t_canq, TTIPRI);
     }
}
```

The second argument to **sleep** (the sleep priority) is set to TTIPRI. This is defined to be 28 (PZERO+3) in *tty.h*, so is an interruptible **sleep**.

NAME

snum – get segment number

SYNOPSIS

```
snum(addr)
unsigned int addr;
```

ARGUMENTS

addr address for which the segment number is to be returned

DESCRIPTION

snum returns the segment number of the specified address.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

segment number

LEVEL

Base or Interrupt

SOURCE FILE

sys//immu.h*

SEE ALSO

pnum(D3X), **poff**(D3X), **psnum**(D3X), **soff**(D3X)

NAME

soff – get segment offset

SYNOPSIS

```
soff(addr)
unsigned int addr;
```

ARGUMENTS

addr address for which the offset is to be returned

DESCRIPTION

soff returns the segment offset of the specified address.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

segment offset

LEVEL

Base or Interrupt

SOURCE FILE

sys//immu.h*

SEE ALSO

pnum(D3X), **poff**(D3X), **psnum**(D3X), **snum**(D3X)

NAME

spl – block/allow interrupts for driver installed under CPU affinity

SYNOPSIS

```
int oldlevel;
oldlevel=spl0();        /* IPL 0; allow all interrupts */
oldlevel=spl1();        /* IPL 1; masks context and process switch */
oldlevel=spl2();        /* IPL 2; blocks all level 1 interrupts */
oldlevel=spl3();        /* IPL 3; blocks all level 1 and level 2 interrupts */
oldlevel=spl4();        /* IPL 4; blocks all level 3 and lower interrupts */
oldlevel=spl5();        /* IPL 5; blocks all level 4 and lower interrupts */
oldlevel=spl6();        /* IPL 6; blocks all level 5 and lower interrupts */
oldlevel=spl7();        /* IPL 7; blocks all interrupts */
oldlevel=splhi();       /* same as spl7 */
oldlevel=spltty();      /* used to protect critical code in TTY drivers */

splx(oldlevel);         /* terminates section of protected critical code */
                        /* and restores interrupt level to previous level */
splx_fast(oldlevel);    /* faster version of splx */
```

ARGUMENTS

oldlevel last set priority value (only **splx** and **splx_fast** have input arguments)

DESCRIPTION

The **spl*** function sets the priority level of the processor on which the code is executing. **splhi** (or other **spl*** function that sets the processor priority level above the level at which the device interrupts) disables interrupts while a section of critical code executes; **splx** or **splx_fast** then restores the processor priority level so that interrupts can be received and handled.

The **spl*** function should not be called directly in drivers installed as fully semaphored. Instead, use semaphores and spin locks to protect resources from unwanted concurrent access. Drivers being ported from other operating systems can be executed without removing the **spl*** code[1] if they are installed under one of the compatibility modes (CPU affinity,[2] major-device semaphoring, or minor-device semaphoring) as described in the *Driver Development Guide*.

spl* is one of the major synchronization functions on traditional UNIX systems, where the system will not switch context from driver code being executed to another executing process unless it is explicitly told to do so by the driver or it receives a device interrupt. By disabling interrupts while executing a piece of critical code (a section of code that updates a shared data structure), the integrity of the kernel is ensured. Because of the

[1] Note that the interrupt latency of drivers installed under major- or minor-device semaphoring can be improved by removing all **spl*** functions from the driver code. The lock on the switch table entry point is adequate to protect critical code sections without the **spl*** functions.

[2] Not all machines support CPU affinity. Refer to the Release Notes shipped with your system.

preemptive kernel and the multiprocessor configuration of the REAL/IX Operating System, the spin lock and semaphore mechanisms are used to protect critical code in fully-semaphored drivers.

The **splx_fast** and **splx** functions restore the interrupt level to the previous level; **splx_fast** is faster than **splx** because it uses the return value of another **spl*** function (such as **splhi**) and does not return the old priority level.

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. 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.

NOTE

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** *or* **splx_fast** *must be used to restore the saved* oldlevel.

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 5, do not call **spl0** *through* **spl4** *or the stack may become corrupted.*

The **spl**–*to-IPL correspondence varies widely from computer to computer. Before executing a ported driver under CPU affinity, it may be necessary to change the values of the* **spl*** *calls to obtain the same interrupt disabling you had on the other machine.*

Drivers that use **spl*** calls must be compiled with **sed**(1) scripts. The *custom/custom.mk* file handles this automatically.

SEMAPHORE RAMIFICATIONS

Drivers that call **spl*** should be installed under one of the compatibility modes.

RETURN VALUE

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

LEVEL

Base or Interrupt

SOURCE FILE

os//interrupt.c*

SEE ALSO

KPG, "Synchronization"
disable(D3X), **enable**(D3X)

NAME

spsema, rspsema, pspsema – lock a spin lock

SYNOPSIS

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

spsema(lock_addr)
lock_t *lock_addr
```

The synopses of **rspsema** and **pspsema** are the same as the synopsis of **spsema**.

ARGUMENTS

lock_addr pointer to a spin lock data structure

DESCRIPTION

The **spsema** family of macros sets a spinning lock on the semaphore specified by *lock_addr* and disables all interrupts. It is appropriate when the lock will be set for a short period of time (less than 50 microseconds); most often, it is used to protect device registers or a region of critical code. Because the stack is used to store old **spl** values, the same routine that sets a spin lock must also unlock that semaphore.

The **rspsema** and **pspsema** macros are faster than **spsema** and can be used to optimize the performance of the driver. **rspsema** can be used if interrupts are already disabled; it is faster than **spsema** because it does not change the **spl** value. **pspsema** can be used if all interrupts are enabled; it is faster than **spsema** because it does not save the **spl** value.

Semaphores locked with one of the **spsema** macros must be unlocked with one of the **svsema** macros in the same routine.

SEMAPHORE RAMIFICATIONS

Drivers that call **spsema** should be installed fully semaphored.

RETURN VALUE

None

LEVEL

Base or Interrupt

SOURCE FILE

sys/sema.h

SEE ALSO

KPG, "Synchronization"
initlock(D3X), **svsema**(D3X), **valulock**(D3X)

NAME

sptalloc – allocate memory pages

SYNOPSIS

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

unsigned int
sptalloc(size, mode, base)
int size, mode, base;
```

ARGUMENTS

size the number of pages to be allocated

mode page descriptor table entry field mask; valid values are:

PG_VALID Indicates that the page descriptor is valid. PG_VALID is defined in *sys/*/immu.h*.

SETCI Specifies that the allocated memory pages will be cache inhibited. The use of SETCI relies on the condition of the flag **badcache**. This flag is set in the kernel if hardware does not maintain cache coherency (e.g., as on the MVME187). Thus, SETCI can be specified only if **badcache** is set.

CAUTION *Specifying SETCI when badcache is not set causes the system to panic.*

base If `base==0`, **sptalloc** allocates physical memory. Otherwise, the value of *base* represents a physical address that is mapped into kernel virtual space.

DESCRIPTION

This function allocates and links virtual memory pages. The normal return value is the kernel virtual address of the allocate space. Allocated space is virtually, but not physically contiguous.

Except for page alignment, using **sptalloc** does not guarantee any alignment of allocated space.

COMPATIBILITY

On some UNIX systems (i.e., other than the REAL/IX Operating System), **sptalloc** takes a fourth parameter, which is a flag indicating whether the function allocating memory can call **sleep**.

Note also that on some UNIX systems, **sptalloc** can use one of several *mode* fields that are not functional on the REAL/IX Operating System.

CAUTION

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

NOTE

In most cases, it is better to use the direct I/O mechanism to move data directly from user address space into the device registers or to allocate memory statically in the driver code.

NOTE

Drivers that allocate memory dynamically are unlikely to be portable.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **sptalloc**.

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)

SOURCE FILE

os/page.c

SEE ALSO

KPG, "Memory Management"
sptfree(D3X)

sptfree(D3X)

NAME

sptfree – free allocated memory

SYNOPSIS

```
sptfree(vaddr, size, mode)
unsigned int vaddr;
int size, flag;
```

ARGUMENTS

vaddr base virtual address of memory to be released

size number of pages to be released

mode must be the same as the *mode* specified in the corresponding call to **sptalloc**(D3X); valid values are:

PG_VALID Indicates that the page descriptor is valid. PG_VALID is defined in *sys/*/immu.h*.

SETCI Specifies that the allocated memory pages will be cache inhibited. The use of SETCI relies on the condition of the flag **badcache**. This flag is set in the kernel if hardware does not maintain cache coherency (e.g., as on the MVME187). Thus, SETCI can be specified only if **badcache** is set.

CAUTION

Specifying SETCI when badcache is not set causes the system to panic.

DESCRIPTION

This function releases memory or performs garbage cleanup to free allocated memory for reuse. This function is called after **sptalloc**(D3X) to free allocated memory.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **sptfree**.

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE *os/page.c*

SEE ALSO *KPG*, "Memory Management"
sptalloc(D3X)

NAME

strcmp, strncmp – compare strings

SYNOPSIS

```
strcmp(s1, s2)
register char *s1, *s2
size_t n;

strncmp(s1, s2, n)
register char *s1, *s2;
```

ARGUMENTS

s1	first string
s2	second string
n	maximum number of characters to compare; used with **strncmp** only

DESCRIPTION

strcmp and **strncmp** are the equivalent of the 3C routines with the same names. They compare two strings and determine if *s1* is lexicographically less than, equal to, or greater than *s2*. **strcmp** evaluates all characters in the string.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

These functions return an integer value that indicates the results of the comparison:

< 0	*s1* is less than *s2*
0	*s1* is equal to *s2*
> 0	*s1* is greater than *s2*

LEVEL

Base or Interrupt

SOURCE FILE

os/string.c

SEE ALSO

string(3C)

NAME

strcpy, strncpy – copy *s2* to *s1*

SYNOPSIS

```
strcpy(s1, s2)
register char *s1, *s2;

strncpy(s1, s2, n)
register char *s1, *s2;
size_t n;
```

ARGUMENTS

s1	destination string
s2	source string
n	number of characters to copy; used with **strncpy** only

DESCRIPTION

strcpy and **strncpy** are the equivalent of the 3C routines with the same names. These functions copy the *s2* string to *s1*. **strcpy** stops only after the null character has been copied; **strncpy** copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. These functions do not check for overflow of the array pointed to by *s1*.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

New value of *s1*.

LEVEL

Base or Interrupt

SOURCE FILE

os/string.c

SEE ALSO

string(3C)

NAME strlen – return length of specified string

SYNOPSIS

```
strlen(s)
char *s;
```

ARGUMENTS

s	string whose length is to be calculated

DESCRIPTION **strlen** is equivalent to the 3C routine with the same name. It returns the number of characters in *s*, not counting the terminating null character.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE The number of characters in *s*.

LEVEL Base or Interrupt

SOURCE FILE *os/string.c*

SEE ALSO **string**(3C)

NAME

subyte – copy a byte from a driver to the user data space

SYNOPSIS

```
subyte(userbuf, c)
caddr_t *userbuf, c;
```

ARGUMENTS

userbuf	address of the user buffer
c	byte to be copied

DESCRIPTION

The **subyte** function copies a byte from the driver 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.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **subyte**.

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)

SOURCE FILE

ml//userio.s*

SEE ALSO

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

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).
- If an error occurs, set **u.u_error** (line 12).

```
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)
7   {
8           if ((c = getc(&tp->t_canq)) == -1) {
9                 return;
10          }

11          if (subyte(u.u_base++, c) == -1) {
12                 u.u_error = EFAULT;
13                 return;
14          }
15          u.u_count--;
16  }
```

NAME suser – verify superuser permission mode

SYNOPSIS `suser();`

ARGUMENTS None.

DESCRIPTION This function determines if the current user has superuser permissions.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE If the current user is a 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)

SOURCE FILE *os/fio.c*

SEE ALSO **rtuser**(D3X), **useracc**(D3X)

EXAMPLE The use of **suser** is straight forward, easy to use, and viable for many situations. The following example shows such a test.

```
if (suser()==0) {
        return;
}
```

On the REAL/IX Operating System, it is more common to check for both realtime privileges and superuser privileges; refer to **rtuser**(D3X) for an example of this use.

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 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.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **suword**.

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)

SOURCE FILE

ml//userio.s*

SEE ALSO

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

EXAMPLE

To debug a driver, a driver **ioctl**(D2X) 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   };

 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) {
20                u.u_rval1 = rp->status;
21
22
23          }else if(suword(arg, rp->status) == -1) {
24
25                u.u_error = EFAULT;
26                return;
27          }
28          break;

29            ⋮

30   }
```

NAME

svsema, rsvsema, psvsema – unlock a spin lock

SYNOPSIS

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

svsema(lock_addr)
lock_t *lockaddr;
```

The synopses for **rsvsema** and **psvsema** are the same as that of **svsema**.

ARGUMENTS

lock_addr identifies the semaphore to be unlocked; must match the *lock_addr* used in the corresponding locking function

DESCRIPTION

The **svsema** family of macros unlocks the spin lock specified by *lock_addr* and sets the interrupt level to the interrupt level that was in effect when the last **spsema** (not **rspsema** or **pspsema**) operation was performed. Because the stack is used to store old SPL values, **svsema** must be called from the same routine that called the locking macro.

rsvsema and **psvsema** perform functionality similar to that of **svsema**, but are faster. **rsvsema** does not modify the interrupt level. **psvsema** sets the interrupt level to have all interrupts enabled.

SEMAPHORE RAMIFICATIONS

Drivers that call **svsema** should be installed fully semaphored.

RETURN VALUE

The **svsema** macros do not return a value under any conditions.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sema.h

SEE ALSO

KPG, "Synchronization"
initlock(D3X), **spsema**(D3X), **valulock**(D3X)

NAME

timeout, timeoutpri, timeoutfs, timeoutfspri – execute a function after a specified length of time

SYNOPSIS

For drivers installed under the compatibility modes:

```
timeout(func, arg, ticks)
int (*func)();
caddr_t arg;
int ticks;
```

For fully-semaphored drivers:

```
timeoutfs(func, arg, ticks)
int (*func)();
caddr_t arg;
int ticks;
```

The parameters for **timeoutpri** are the same as for **timeout**; the parameters for **timeoutfspri** are the same as for **timeoutfs**.

ARGUMENTS

func	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** family of functions calls the specified function after a specified time interval. After the specified number of clock ticks, the function specified by *func* is invoked with all interrupts disabled; the function should reenable interrupts by invoking **enable**(D3X) at the earliest possible opportunity. Control is returned immediately to the caller.

The timeout functions are 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 one of the **timeout** functions, the driver can wait a predetermined interval and then begin transferring data to the robot.

The system guarantees that the time that elapses between the call to **timeout** and the execution of *func* is not less than the value specified by *ticks*. The function is scheduled *ticks* after the next clock tick; thus, the average delay typically is half a clock tick more than was requested. Note also that other processing may cause the execution of *func* to take place some time after it was scheduled. The delay is given in terms of a notional system clock that ticks at a rate determined by the constant HZ, which is defined in the *param.h* header file (the actual tick rate of the system clock may be higher than the value of HZ).

When the specified time has elapsed, the system arranges for the user-defined function *func* to be called. The function is actually called from a system daemon. The daemon is responsible for servicing other timer functions, which means *func* cannot be allowed to block.[1] For these reasons, *func* 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. Furthermore, *func* should not call **sleep**(D3X), **delay**(D3X), or **psema**(D3X). However, in a fully-semaphored driver, data in *func* can be protected, if necessary, with spin locks (**spsema**(D3X) and **svsema**(D3X)).

When called from a driver using major- or minor-device semaphoring, the semaphore used for **timeout** or **timeoutpri** is recorded in the kernel data structure that controls the timeout. When the **timeout** period expires, an attempt is made to lock the driver semaphore before calling the specified function. If the lock attempt fails, the entry will be processed again on the next clock interrupt.

SEMAPHORE RAMIFICATIONS

Drivers that call **timeout** or **timeoutpri** must be installed under the compatibility modes.

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 size of the table is determined by the **sysgen** parameter NCALL. The default setting should be sufficient for all but the most unusual configuration.

[1] System daemons typically operate at high priorities. For timeout processing to work correctly, the priority of the daemon handling a particular timeout must be higher than the priority of the initiating process. Therefore, there must be at least one such daemon at very high priority, usually at priority 0. The **timeout** and **timoutfs** calls implicitly request the use of this high-priority daemon. The **timeoutpri** and **timeoutfspri** calls are for use only from the base level of a process; these functions allow the REAL/IX Operating System to examine the priority of the calling process and to arrange for a daemon of appropriate priority to handle the timeout processing. The use of **timeoutpri** and **timeoutfspri** is preferred.

All the **timeout** functions return an identifier that can be passed to the **untimeout**(D3X) function to cancel a pending request.

Note that no value is returned from the called function.

LEVEL

For **timeout** and **timeoutfs** – Base or Interrupt
For **timeoutpri** and **timeoutfspri** – Base only

SOURCE FILE

os/clock.c

SEE ALSO

KPG, "Synchronization"
delay/delayfs(D3X), **iodone**(D3X), **iowait**(D3X), **sleep**(D3X), **spsema**(D3X), **svsema**(D3X), **ttywait**(D3X), **untimeout**(D3X), **wakeup**(D3X)

EXAMPLE

Refer to the **untimeout**(D3X) examples for an example of how to call **timeout** family of functions.

NAME

ttclose – close a TTY device

SYNOPSIS

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

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

ARGUMENTS

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 output queues (tp->t_outq and tp->t_buf) 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.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttclose** must be installed under the compatibility modes.

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/tt1.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
ttopen(D3X)

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)];

6        (*linesw[tp->t_line}.l_close)(tp);

7           ⋮
```

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, then calls **ttyflush**(D3X) 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.

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.[1] When the raw queue character count exceeds the TTYHOG level, **ttin** calls **ttyflush** to flush the tty input queue. 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 processes.

The **ttin** function also transmits characters to the terminal for display, if ECHO is enabled.

When the terminal operates in a 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 length 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.

[1] The high water mark is the point at which data being processed in the output queue of a clist(D4X) is transmitted to the terminal.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttin** must be installed under one of the compatibility modes.[1]

RETURN VALUE None

LEVEL Base or Interrupt

SOURCE FILE *io/tt1.c*

SEE ALSO *KPG*, "Drivers in the TTY Subsystem"
getc(D3X), **getcb**(D3X), **getcf**(D3X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **ttread**(D3X)

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] Not all compatibility modes are supported on all machines. Refer to the Release Notes shipped with your system.

```
struct device                          /* Layout of physical device register */
{
              int   control;           /* Physical device control word */
              int   status;            /* Physical device status word */
              short recv_char;         /* Receive character from device */
              short xmit_char;         /* Transmit character to device */
};                                     /* End device */

extern struct tty    xx_tty[];         /* Logical device structure location */
extern struct device xx_addr[];        /* Physical device register location */
extern int           xx_cnt;           /* Number of physical devices */

    ⋮

xx_rint(board)
int board;                           /* The hardware board causing interrupt */
{
register struct device *rp = xx_adddr[board];  /* Get device registers */
register struct tty *tp;
register int c, port;

while((c = rp->recv_char) & DATAVALID) != 0)
{
      port = (c >> 8) & 0x7;
      tp = &xx_tty[(board << 3) & port];

/* After the character has been checked for errors and stripped to */
/* proper bit size, character is stored in receive buffer.  */

      *tp->t_rbuf.c_ptr++ = c;
      if (--tp->t_rbuf.c_count == 0)
      {
          /* driver must do operation to ensure the buffer added  */
          tp->t_rbuf.c_ptr -= tp->t_rbuf.c_size; /* to raw queue correctly */

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

      }
}

    ⋮
```

NAME

ttinit – initialize line discipline 0

SYNOPSIS

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

ttinit(tp)
struct tty *ty;
```

ARGUMENTS

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**, **t_lflag** members of the tty data structure. It also sets the default control modes (**t_cflag**) and control characters (**t_cc**), and sets **t_rsel** and **t_wsel** to 0 for **select**(D2X).

NOTE

***ttinit** is usable only for resetting line discipline 0. Using **ttinit** on any other line discipline requires resetting **t_line** to a new value after **ttinit** is called.*

SEMAPHORE RAMIFICATIONS

Drivers calling **ttinit** must be installed under the compatibility modes.

RETURN VALUE

None

LEVEL

Base or Interrupt

SOURCE FILE

io/tty.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
open(D2X), **ttopen**(D3X)

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 9600 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)
12     {
13        ttinit(tp);
14        tp->t_proc = xx_proc;
15
16     /* The appropriate device registers would be set to match the */
17     /* values stored in the tty structure - hardware dependent. */
18      } /* endif */

19        ⋮
```

NAME

ttiocom – common **ioctl** code for TTY drivers

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 about the commands described here

Valid commands (listed in alphabetic order) are

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*.
TCSBRK	Waits for the output to drain. If *arg* is 0, then sends a BREAK character
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 the subordinate form of a command that 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 requires that device registers also be altered. The driver is responsible for changing these registers.

A request to read or change terminal parameters is initiated by an **ioctl**(2) system call from a user process. This causes the driver **ioctl**(D2X) routine to be called. The driver locates the `tty` structure associated with the device and calls the common **ioctl** routine **ttiocom**.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttiocom** must be installed under the compatibility modes.

RETURN VALUE

Under normal conditions, 0 (zero) is returned. Otherwise, 1 is returned to indicate 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 for the specified commands:

  - ▪ TCGETA **copyout** failed
  - ▪ TCSETA **copyin** failed

- ❑ EINVAL invalid argument. This value is set under the following conditions for the specified commands:

  - ▪ TCFLSH *arg* not in the range of 0 to 2
  - ▪ TCSETA line discipline value in the **c_line** member of the **termio** structure not 0
  - ▪ TCXONC *arg* not in the range of 0 to 3

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/tty.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
ioctl(D2X), **ttioctl**(D3X)

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 8).
- The **switch** statement (line 9) should contain **cases** that handle driver-specific commands, such as getting the device registers.
- The **default** is to handle **termio**(7) commands. If an invalid command is present, **ttiocom** will update **u.u_error** with EINVAL.
- If changes are made in the parameters of the `tty` structure (line 13), then the device registers may also need to be altered (lines 14 and 15); the driver would make the necessary changes upon return from the **ttiocom** function.

  Changes are usually determined by examining the parameter settings in the **t_iflag**, **t_oflag**, **t_cflag**, and **t_lflag** members of the `tty`(D4X) structure for changes such as baud rate, parity type, testing, and so forth. These values are hardware dependent.

The line discipline switch table is **not** to be used for a line discipline 0 **ioctl** request.

```
extern struct device xx_addr[];        /* Physical device register location */
extern struct tty  xx_tty[];           /* logical device structure location */

    ⋮

xx_ioctl(dev, cmd, arg, flag)
dev_t  dev;
caddr_t arg;
{
        register struct tty *tp = &xx_tty[minor(dev)]; /* Get tty structure */
        switch(cmd) {
             case statements for driver-specific commands
        default:
             handle termio(7) commands

             if (ttiocom(tp, cmd, arg, flag) == 1) {
                     register struct device *rp;
                     rp = &xx_addr[minor(dev) >> 3];   /* Get device regs  */
             }
        }
}
```

NAME

ttioctl – default line discipline **ioctl** routine

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** *cmds* 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. For drivers that use **ttyd** (the tty daemon), it then allocates another cblock for the raw input buffer (**t_ribuf**).

LDCLOSE resume output by calling the driver **proc**(D2) routine with the T_RESUME command, wait for all characters remaining in the output queue to drain, 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** calling the **t_flag** member of the tty structure.

arg flag indicates the subordinate form of a command that 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.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttioctl** must be installed under the compatibility modes.

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/tt1.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
ioctl(D2X), **ttiocom**(D3X)

NAME

ttopen – open a TTY device

SYNOPSIS

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

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

ARGUMENTS

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**), then calls **ttioctl** with the LDOPEN command, which calls the driver **proc**(D2X) routine with T_INPUT set.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttopen** must be installed under the compatibility modes.

RETURN VALUE

None. **ttopen** sets **t_state** to ISOPEN.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/tt1.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
linesw(D4X), **open**(D2X), **ttclose**(D3X), **ttinit**(D3X)

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 (refer to **ttinit**(D3X)), 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 operation in the driver's **open** routine establishes a logical data connection and associates the device with a process by making the appropriate settings in the tty structure (line 34).

□ 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.

```
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   };

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  {
16  register struct tty *tp = &xx_tty[minor(dev)];
17  register struct device *rp = &xx_addr[minor(dev) >> 3];
                                                         /* Get device regs */
18    ⋮

19
20   if ((rp->modem_status & (0x010 << port)) != 0) {
22                tp->t_state |= CARR_ON;
23         } else {
24                tp->t_state &= ~CARR_ON;
25         }

26         if ((flag & FNDELAY) == 0) {
27              while((tp->t_state & CARR_ON) == 0) {
29                        tp->t_state |= WOPEN;
30                        sleep((caddr_t)&tp->t_canq, TTIPRI);
31              }
32         }
33  }
34  (*linesw[tp->t_line].l_open)(tp);
```

NAME

ttout - move TTY characters from **t_outq** to **t_tbuf**

SYNOPSIS

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

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

ARGUMENTS

tp pointer to the tty(D4X) structure associated with the device

DESCRIPTION

The **ttout** function is called by the transmit portion of the driver's **intr**(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 a restart of the output after the appropriate time has elapsed. This delayed entry invokes the driver **proc**(D2X) routine with T_TIME set to resume output.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttout** must be installed under the compatibility modes.

RETURN VALUE

Under normal conditions, 0 (zero) is returned when there is no more data to process. CPRES is returned if there are characters in the output queue. (CPRES is set to octal 100000 in *tty.h*).

LEVEL

Base or Interrupt

SOURCE FILE

io/tt1.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
linesw(D4X), **ttin**(D3X)

NAME

ttread – read characters from the canonical input queue

SYNOPSIS

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

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

ARGUMENTS

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**.

ttread transfers data from the canonical input queue into user data space. If there are no characters in the canonical queue, an attempt is made to move characters into the canonical from the raw input queue. If there are still no characters available to be read, the calling process is put to sleep until sufficient characters arrive to satisfy the read, or the read times out via the VTIME option (termio(7)). If input to the raw queue was previously blocked (**t_state & T_BLOCK**) and the number of characters in the raw queue falls below the low water mark, **ttread** calls the driver's **proc**(D2X) routine with T_UNBLOCK to allow input into the raw queue to continue.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttread** must be installed under the compatibility modes.

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)

SOURCE FILE

io/tt1.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
getc(D3X), **getcb**(D3X), **getcf**(D3X), linesw(D4X), **putc**(D3X), **putcb**(D3X), **putcf**(D3X), **read**(D2X), **ttin**(D3X)

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);
8   }
```

NAME

ttrstrt – restart TTY output after delay timeout

SYNOPSIS

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

ARGUMENTS

tp	pointer to the tty(D4X) structure

DESCRIPTION

This function restarts TTY output following a delay timeout. **ttrstrt** calls the driver **proc**(D2X) routine with the T_TIME command.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttrstrt** must be installed under the compatibility modes.

RETURN VALUE

None

LEVEL

Base or Interrupt

SOURCE FILE

io/tty.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
timeout(D3X)

EXAMPLE

When a TCSBRK command is issued in an **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 through 67) for the code for the T_OUTPUT case that is shown as comments in lines 29 and 30 of this example.

```
struct device                     /* Layout of physical device registers */
{
      int   control:              /* Physical device control word */
      int   status;               /* Physical device status word */
      short modem_status;         /* Modem carrier (upper 8 bits) */
                                  /* and ring (lower 8 bits) status word */
      short recv_char;            /* Receive character from device */
      short xmit_char;            /* Transmit character to device */
};
extern struct device xx_addr[];  /* Physical device registers */
extern struct tty  xx_tty[];     /* Logical device structures location */

  ⋮

xx_proc(tp, cmd)                   /* Driver command processing routine */
register struct tty *tp;
int cmd;
{
register int  dev = tp - xx_tty;      /* Compute minor device number */
register struct device *rp = &xx_addr[dev >> 3];  /* Get device regs */
register int  portmask = 0x0100 << (dev & 0x7);
 /* Set up output port mask */
switch(cmd)
{
case T_TIME:
   tp->t_state &= ~TIMEOUT;
   goto resume_output;           /* Resume normal character output */

      ⋮

case T_OUTPUT: /* Perform output processing of data to the device */
resume_output:
         /* Transmit next tbuf character of the tty structure */
         /* Refer to ttout(D3X) for example program code */
   break;

      ⋮

case T_BREAK:
   rp->control |= XX_BRK;
   rp->xmit_char |= portmask;
   tp->t_state |= TIMEOUT;
   timeout(ttrstrt, tp, HZ/4);  /* Disable timeout condition 1/4 of */
                                /* a second (HZ) or 250 milliseconds */
   break;

      ⋮
```

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;
```

ARGUMENTS

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** taw input buffer.

SEMAPHORE RAMIFICATIONS

Drivers calling **tttimeo** must be installed under the compatibility modes.

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

SOURCE FILE

io/tt1.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
canon(D3X), **timeout**(D3X)

EXAMPLE

The following example shows the use of **tttimeo** (line 15) 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 to outq if
 7         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             }
13             else if (tp->t_cc[VTIME]) {
14                    if(!(tp->t_state&TACT))
15                          tttimeo(tp);
16             }
17        }
18   }
```

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;
```

ARGUMENTS

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 because 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's 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 number to determine the tty structure for the device being written. The address of this structure 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.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttwrite** must be installed under the compatibility modes.

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)

SOURCE FILE

io/tt1.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
linesw(D4X)

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   }
```

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 character 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 0200 (octal), if the character is a delayed character. When processing 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.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttxput** must be installed under the compatibility modes.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

io/tt1.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
ttin(D3X), **ttwrite**(D3X)

EXAMPLE

The following example uses **ttxput** (line 13) 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   c = *tp->t_rbuf.c_ptr++;
 9   /* transfer characters from t_rbuf to t_rawq performing input
10   translation if necessary */
11          if (flg & ECHO) {
12                 /* place character - 'c' - on t_outq */
13                 ttxput(tp, c, 0);
14                 /* initiate physical output */
15                 (*tp->t_proc)(tp, T_OUTPUT);
16          }
17   /* check to see if non-canonical timing should be done */
18   }
```

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**

SEMAPHORE RAMIFICATIONS

Drivers calling **ttyflush** must be installed under the compatibility modes.

RETURN VALUE

None

LEVEL

Base or Interrupt

SOURCE FILE

io/tty.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
cblock(D4X), **clrbuf**(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;
```

ARGUMENTS

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 serial device is drained.

SEMAPHORE RAMIFICATIONS

Drivers calling **ttywait** must be installed under the compatibility modes.

RETURN VALUE

None

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/tty.c

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
delay(D3X), **iodone**(D3X), **iowait**(D3X), **sleep**(D3X), **timeout**(D3X), **untimeout**(D3X), **wakeup**(D3X)

undma(D3X)

NAME

undma – unlock memory locked with **userdma**(D3X)

SYNOPSIS

```
undma(base, count, rw)
int base, count, rw;
```

ARGUMENTS

All arguments must match exactly the arguments used with the corresponding **userdma** call.

base the start address of the user data area

count the size of the data transfer, in bytes

rw flags to determine whether the access is a read or write operation and whether to lock down the memory. Refer to **userdma**(D3X) for the valid values.

DESCRIPTION

undma reverses the effect of **userdma**(D3X).

CAUTION

undma *assumes that the parameters it is given are exactly as per the original call to* **userdma**. *In any case, it has no ready means by which to validate them. Passing incorrect parameters to the* **undma** *function will give undefined and potentially catastrophic results.*

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **undma**.

RETURN VALUE

None.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/probe.c

SEE ALSO

klock(D3X), **kunlock**(D3X), **useracc**(D3X), **userdma**(D3X)

NAME

untimeout – cancel prior **timeout/timeoutfs/timeoutpri/timeoutfspri**(D3X) function call

SYNOPSIS

```
untimeout(id)
int id;
```

ARGUMENTS

id identification value generated by a previous **timeout/timeoutfs** function call

DESCRIPTION

The **untimeout** function cancels a pending **timeout** request.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

None.

LEVEL

Base or Interrupt

SOURCE FILE

os/clock.c

SEE ALSO

KPG, "Synchronization"
DELAY(D3X), **delay/delayfs**(D3X),
timeout/timeoutfs/timeoutpri/timeoutfspri(D3X), **ttywait**(D3X)

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 queued, the driver may want to cancel the **timeout** operation before it expires. 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 online, a message is displayed on the system console. The driver schedules a wakeup call (line 41) and waits for 5 minutes. 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). There is also code (lines 42 through 48) to cancel the **timeout** if the process that is sleeping while waiting for the device receives a signal. In this case, cleanup is effected by canceling the pending **timeout** request and issuing a **klongjmp**(D3X) to return.

```
struct mtu_device               /* Layout of physical device registers */
{
      int     control;          /* Physical device control word */
      int     status;           /* Physical device status word */
      int     byte_cnt;         /* Number of bytes to be transferred */
      paddr_t baddr;            /* DMA starting physical address */
};                              /* end device */

struct mtu                      /* Magnetic tape unit logical structure */
{
      struct buf *mtu_head;     /* Pointer to I/O queue head */
      struct buf *mtu_tail;     /* Pointer to buffer I/O queue tail */
      int    mtu_flag;          /* Logical status flag */
      int    mtu_to_id;         /* Time out id number */

        :

};                              /* end mtu */

extern struct mtu_device *mtu_addr[];  /* Location of device registers */
extern struct mtu  mtu_tbl[];          /* Location of device structures */
extern int    mtu_cnt;

  :

mtu_open(dev, flag)
dev_t dev;
{
register struct mtu *dp;
register struct mtu_device *rp;
   if ((minor(dev)>> 3) > mtu_cnt) {  /* If device does not exist, */
      u.u_error = ENXIO;              /* then return error condition */
      return;
   }                                  /* endif */
   dp = &mtu_tbl[minor(dev)];             /* Get logical device struct */
   if (dp->mtu_flag & MTU_BUSY) != 0) {   /* If device is in use, */
      u.u_error = EBUSY;                  /* return busy status */
      return;
   }                                      /* endif */
  dp->mtu_flag = MTU_BUSY;          /* Indicate device in use & clear flags */
  rp - xx_addr[minor(dev) >> 3];    /* Get device regs */
  oldlevel2 = splhi();

 /* While tape not loaded, display mount request on console */

 while((rp->status & MTU_LOAD) == 0) {
   cmn_err(CE_NOTE, "Tape MOUNT request for driver %d", minor(dev) & 0x3);
   dp->mtu_flag |= MTU_WAIT;        /* Indicate process suspended */
   dp->mtu_to_id = timeoutpri(wakeup, dp, 5*60*HZ);   /* Wait 5 min */
   /* Wait on tape load. If user aborts process,
     release tape device by clearing flags */
   if (sleep(dp, (PCATCH | PZERO + 2)) == 1) {
       dp->mtu_flag = 0;
       untimeout(dp->mtu_to_id);
       splx_fast(oldlevel2);
       klongjmp();        /* Abort open(2) system call */
   }
 }                                  /* end while */
  splx(oldlevel2);
}

  :

mtu_int(cntr)
int cntr;                   /* Controller that caused the interrupt */
{
register struct mtu_device *rp = xx_addr[cntr];   /* Get device regs */
register struct mtu *fp = &mtu_tbl[cntr >> 3 | (rp->status & 0x3)];

  :

/* If process is suspended waiting for tape mount, */
if ((dp->mtu_flag & MTU_WAIT) != 0) {
    untimeout(dp->mtu_to_id);          /* cancel timeout request */
    dp->flag &= ~MTU_WAIT;             /* Clear wait flag  */
    wakeup(dp);                        /* Awaken suspend process */
}

  :
```

NAME

upath – copy data from user space to kernel space

SYNOPSIS

```
upath(userbuf, kernelbuf, maxbufsz)
caddr_t userbuf, kernelbuf;
int maxbufsz;
```

ARGUMENTS

userbuf user program source address from which data is transferred

kernelbuf kernel destination address to which data is transferred

maxbufsz maximum number of bytes to move (determined by buffer that was allocated)

DESCRIPTION

The **upath** function copies data from a user process to a kernel process. It is similar to **copyin**(D3X), except that **copyin** moves the specified number of bytes, whereas **upath** copies until it encounters a NULL character (the NULL is copied) or reaches the number of bytes specified by *maxbufsz*.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **upath**.

RETURN VALUE

If successful, **upath** returns the number of bytes copied, not including the NULL. Otherwise, it returns one of the following:

- ❑ –1 indicates a paging fault (the driver tried to access a page of memory for which it did not have read access); the driver should set the **u.u_error** member of user(D4X) to EFAULT.
- ❑ –2 indicates that no NULL character was found; the driver should set the **u.u_error** member of user(D4X) to E2BIG.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

ml//userio.s*

SEE ALSO

copyin(D3X)

EXAMPLE

The following code illustrates how **upath** is called:

```
len = upath((caddr_t)ap, vaddr, cc);
if (len == -1) {
     u.u_error = EFAULT;
     return;
}
if (len == -2) {
     u.u_error = E2BIG;
     return;
}
```

NAME

useracc – verify whether user has access to memory

SYNOPSIS

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

int
useracc(base, count, access)
int base;
int count, access;
```

ARGUMENTS

base the start address of the user data area (typically taken from 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).

rw flags to determine whether the access is a read or write operation, and whether or not to lock down the memory. Valid values are:

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.

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*).

B_PHYS causes the user virtual memory (described by *base* and *count*) to be faulted, if necessary, and then locked. This guarantees that the buffer will not be paged out during the I/O transfer.

SEMAPHORE RAMIFICATIONS

No spin locks should be held when calling **useracc**.

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).

Note that, when used with the B_PHYS flag, **useracc** is equivalent to the **userdma**(D3X) function.

RETURN VALUE

If successful, **useracc** returns 1. Otherwise, 0 (zero) is returned and an error code is set in **u.u_error**. Possible errors are:

EAGAIN	Insufficient kernel resources to lock page.
EFAULT	B_READ is set, but the memory is marked as being read-only (a read from a device has to write to memory, which is not allowed).
EFAULT	The memory described by *base* and *count* is not within the user's address space.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os/probe.c

SEE ALSO

klock(D3X), **kunlock**(D3X), **rtuser**(D3X), **suser**(D3X), **undma**(D3X), **userdma**(D3X)

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.

- ❑ *nblks* defines the blocks to be read (line 8) or written (line 37) with direct I/O (**physio**(D3X)) to or from a block device. The data must be moved as a single complete block or multiples of complete blocks

- ❑ For a read request, a test is made to determine if the I/O request is in the limits of the RAM disk (line 12) and, if so, the driver computes the number of blocks that can be copied (line 14).

- ❑ For a write request, a test is made to ensure that there are one or more complete blocks to be copied (line 41). If not, the driver sets **u.u_error** to EFAULT (line 45).

- ❑ With a demand paging system, the driver must ensure that the user's program data pages are in memory by calling **useracc** (lines 19 and 48). If an error occurs, **useracc** will set **u.u_error** to an error code; the driver does not need to do it.

- ❑ The driver then computes the starting block number and copies the data to the user (lines 25 through 30 and lines 54 through 59).

This example is based on an example in the AT&T documents. Although it is valid on the REAL/IX Operating System, the use of **useracc** with **copyin** and **copyout** is redundant because those functions handle any page faults that might occur.

```
 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)
 5   dev_t dev;
 6   {
 7   register daddr_t blkno;                   /* Starting block number */
 8   register int     nblks;                   /* Number of logical blocks */
14      if (u.u_count % RAMDBSIZ)  {
16                 u.u_error = EFAULT;
17                 return;
18      }
14      if (u.u_offset % RAMDBSIZ)  {
16                 u.u_error = EFAULT;
17                 return;
18      }
12      if (physck(RAMDNBLK,B_READ)) {
19           if (useracc(u.u_base, u.u_count, B_READ) == 0) {
23                  return;
24           }
25           blkno = u.u_offset % RAMDBSIZ;
27           copyout (u.u_base, (caddr_t)&ramdblks[blkno][0], u.u_count);
28           u.u_base += u.u_count;      /* Increment virtual base addr */
29           u.u_offset += u.u_count;    /* Increment file offset */
30           u.u_count = 0;              /* No more bytes to be transferred */
31      }
32   }
33   ramdwrite(dev)
34   dev_t dev;
35   {
36   register daddr_t blkno;      /* Starting block number */
37   register int     nblks;      /* Number of logical blocks to be written */
43      if (u.u_count % RAMDBSIZ != 0) {
45           u.u_error = EFAULT;
46           return;
47      }
43      if (u.u_offset % RAMDBSIZ != 0) {
45           u.u_error = EFAULT;
46           return;
47      }
```

```
    if (physck(RAMDNBLK,B_WRITE)) {
         if(useracc(u.u_base, u.u_count, B_WRITE) == 0) {
                return;
         }
         blkno = u.u_offset / RAMDBSIZ;
         copyin (u.u_base, (caddr_t)&ramdblks[blkno][0], u.u_count);
         u.u_base += u.u_count;         /* Increment virtual base addr */
         u.u_offset += u.u_count;       /* Increment file offset */
         u.u_count = 0;                 /* No more bytes to be transferred */
    }
}
```

NAME

userdma – lock user virtual memory for DMA transfer

SYNOPSIS

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

userdma(base, count, rw)
int base, count, rw;
```

ARGUMENTS

base the start address of the user data area (typically taken from 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).

rw flags to determine whether the access is a read or write operation, and whether or not to lock down the memory. Valid values are:

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.

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

The **userdma** function causes the area of user virtual memory described by *base* and *count* to be faulted if necessary and then locked. This guarantees that the buffer will not be paged out during the I/O operation.

userdma is equivalent to **useracc**(D3X) with the B_PHYS access flag.

SEMAPHORE RAMIFICATIONS

No semaphores or spin locks should be held when calling **userdma**.

RETURN VALUE

If successful, **userdma** returns 1. Otherwise, 0 (zero) is returned and an error code is set in **u.u_error**. Possible errors are:

EAGAIN	Insufficient kernel resources to lock page.
EFAULT	B_READ is set, but the memory is marked as being read-only (a read from a device has to write to memory, which is not allowed).

EFAULT The memory described by *base* and *count* is not within the user's address space.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

sys/klock.h

SEE ALSO

dma_breakup(D3X), **physck**(D3X), **physio**(D3X), **undma**(D3X), **useracc**(D3X)

EXAMPLE

The following example illustrates the use of **userdma**.

```
if (userdma(base, count, rw) == NULL) {
     if (u.u_error == 0)
          u.u_error = EFAULT;
     return;
}
```

NAME

VMEbus

usshmctl – install user-defined special shared memory control function into the kernel

SYNOPSIS

```
int usshmctl(sshmtype, func)
uint sshmtype;
int (*func) ();
```

ARGUMENTS

sshmtype number of the user special shared memory type; must be in the range of 8 through 15

func name of the special shared memory control function

DESCRIPTION

usshmctl installs the control function of a user-defined special shared memory type into the kernel. **usshmctl** must be called for each user-defined special shared memory type. If multiple user-defined special shared memory types are defined, the corresponding type numbers must be selected sequentially starting with 8. By convention, all calls to the **usshmctl** function are coded in the *usysinit.c* file in the */usr/src/uts/realix/custom* directory.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

If successful, **usshmctl** returns 0. Otherwise, a −1 is returned and an error is written to the console and */usr/adm/putbuf*.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

io/vme/sshm.c

SEE ALSO

KPG, "Miscellaneous I/O Operations"

EXAMPLE

This example shows the *usysinit.c* file with a special shared memory control function (sshmctlmeg) defined. The user-defined special shared memory type number is 8.

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

extern int sshmctlmeg();

int
usysinit()
{
     usshmctl(8, sshmctlmeg);
}
```

NAME

VMEbus

usyscall – install user-defined system call into the kernel

SYNOPSIS

```
int usyscall(nsyscall, func, nargs)
unsigned int nsyscall, nargs;
int (*func) ();
```

ARGUMENTS

nsyscall number of the system call in the `sysent` table, usually expressed in terms of USYSCALLOW (lowest allowed value) and USYSCALLHI (highest allowed value)

func name of the system call

nargs number of arguments for the system call

DESCRIPTION

usyscall installs a user-defined system call into the kernel. By convention, **usyscall** functions for all user-defined system calls are coded in the *usysinit.c* file in the */usr/src/uts/realix/custom* directory.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

If successful, **usyscall** returns 0. Otherwise, a −1 is returned and an error is written to the console and */usr/adm/putbuf*.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

os//sysent.c*

SEE ALSO

KPG, "Writing and Installing System Calls"

EXAMPLE

This example shows the *usysinit.c* file with two system calls defined. The first system call definition is for the first available user entry in the `sysent` table, which is called **respages** and has one argument; the second one is for the second available user entry in the `sysent` table, which is called **mycall** and has three arguments.

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

extern int respages();

int
usysinit()
{
        usyscall(USYSCALLOW, respages, 1);
        usyscall(USYSCALLOW+1, mycall, 3;
}
```

NAME

uvtopde – return page descriptor entry for user virtual address

SYNOPSIS

```
pde_t *
uvtopde(uva)
unsigned int uva
```

ARGUMENTS

uva	user virtual address

DESCRIPTION

This macro returns the address of the page descriptor entry that maps the user virtual address for the process.

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

The physical address of the page table entry.

LEVEL

Base Only (Do not call from an interrupt routine)

SOURCE FILE

sys/immu.h or *cf/inlines/sed**

NAME

valulock – return current value of a spin lock

SYNOPSIS

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

val = valulock(lock_addr);
lock_t *lock_addr;
```

ARGUMENTS

lock_addr the spin lock being checked; must match the *lock_addr* used when the spin lock was initialized with the **initlock** macro

DESCRIPTION

The **valulock** macro returns the current value of the spin lock specified by *lock_addr*.

SEMAPHORE RAMIFICATIONS

Drivers that call **valulock** must be installed fully semaphored.

RETURN VALUE

valulock returns the current value of the spin lock. 0 indicates that the resource is not currently locked. 1 indicates that the resource is currently locked.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sema.h

SEE ALSO

KPG, "Synchronization"
spsema(D3X), **svsema**(D3X), **initlock**(D3X)

NAME

valusema – return current value of a semaphore

SYNOPSIS

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

val = valusema(sem_addr);
sema_t *sem_addr;
```

ARGUMENTS

sem_addr the semaphore being checked; must match the *sem_addr* used when the semaphore was initialized with the **initsema** or **reinitsema** macros

DESCRIPTION

The **valusema** macro returns the current value of the semaphore specified by *sem_addr*.

SEMAPHORE RAMIFICATIONS

Drivers that call **valusema** should be installed fully semaphored.

RETURN VALUE

valusema returns the current value of the semaphore:

- ❑ 1 or >1 indicates that the resource is not currently locked.
- ❑ 0 indicates that the resource is currently locked and no other processes are blocked waiting for the resource.
- ❑ <0 indicates that the resource is locked and other processes are blocked waiting for the resource. The absolute value of the value returned is the number of processes waiting for the resource.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sema.h

SEE ALSO

KPG, "Synchronization"
cpsema(D3X), **cvsema**(D3X), **decsema**(D3X), **incsema**(D3X), **initsema**(D3X), **psema**(D3X), **psvsema**(D3X), **vsema**(D3X)

NAME

VMEbus

vme_a24_mem_valid – verify that an address is accessible by A24 VME devices

SYNOPSIS

```
vme_a24_mem_valid(paddr, bufsiz)
unsigned int paddr, bufsiz
```

ARGUMENTS

paddr physical address, usually obtained through **disjointio**(D3X) or the kernel-virtual-to-physical macro

bufsiz the size of the buffer

DESCRIPTION

This macro determines if the buffer described is within A24 address space (in other words, that *paddr* + *bufsiz* is less than or equal to 8 megabytes).

SEMAPHORE RAMIFICATIONS

None.

RETURN VALUE

1 if the entire range from *paddr* to *paddr+siz−1* resides in A24 address space.

0 if any portion of the range is outside A24 space.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sysmacros.h

SEE ALSO

KPG, "Memory Management"

NAME

vsema, rvsema, pvsema – unlock semaphore for a resource or make resource available

SYNOPSIS

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

val = vsema(sem_addr, reserved, flags);
sem_t *sem_addr;
int *reserved;
int flags;
```

The synopses for **rvsema** and **pvsema** are the same as the synopsis of **vsema**.

ARGUMENTS

sem_addr identifies the semaphore to be unlocked; must correspond to the *sem_id* used to lock the resource

reserved the second argument is reserved for future use; in this release, it must always be 0

flags flag parameter; valid values are:

0	Used when the run queue lock is not currently locked and the semaphore is not one for which a boosting algorithm is defined.
SEMRTBOOST	Used if the corresponding **psema** used the SEMRTBOOST flag. No other flags can be used.

DESCRIPTION

The **vsema** family of functions increments the value of the semaphore specified by *sem_addr*. If the value of the semaphore was negative (indicating that a process was blocked on the semaphore), **vsema** unblocks the first process (the process with the highest priority) on the list of processes that were blocked after doing a **psema** on the semaphore.

rvsema and **pvsema** perform functionality similar to that of **vsema**, but are faster. **rvsema** can be used when all interrupts are disabled; **pvsema** can be used when all interrupts are guaranteed to be enabled.

SEMAPHORE RAMIFICATIONS

Drivers that call **vsema** must be installed fully semaphored.

RETURN VALUE

The **vsema** macros do not return a value under any conditions.

LEVEL

Base or Interrupt

SOURCE FILE

sys/sema.h

SEE ALSO

KPG, "Synchronization"
cpsema(D3X), **cvsema**(D3X), **psema**(D3X), **psvsema**(D3X), **initsema**(D3X), **valusema**(D3X)

NAME

wakeup – resume unsuspended process execution

SYNOPSIS

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

wakeup(addr)
caddr_t addr;
```

ARGUMENTS

addr address on which process is sleeping (corresponds to *addr* used with **sleep**(D3X)

DESCRIPTION

The **wakeup** function awakens all processes that called **sleep** with this *addr* argument. This lets the processes execute according to the scheduler. You must use the same *addr* for both **sleep** and **wakeup**. For code readability and efficiency, it is best 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 that the **sleep**(*addr*) occurred. There is an interval between the time the process that called **sleep** is awakened and the time it resumes execution when 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**.

The **wakeup** function can be used in REAL/IX drivers only if the driver is installed under CPU affinity[1] or major- or minor-device semaphoring. Drivers that are fully semaphored use spin locks and semaphores to provide sleep/wakeup synchronization.

Note that a driver that calls **sleep** and **wakeup** should not call **psema**, **cpsema**, or **vsema**, and vice versa. Mixing the sort of synchronization done in one driver will result in deadlocks.

SEMAPHORE RAMIFICATIONS

Drivers calling **wakeup** must be installed under the compatibility modes.

RETURN VALUE

None

LEVEL

Base or Interrupt

[1] Not all machines support CPU affinity. Refer to the Release Notes shipped with your system.

SOURCE FILE *os/slp.c*

SEE ALSO *KPG*, "Synchronization"
delay(D3X), **iodone**(D3X), **iowait**(D3X), **sleep**(D3X), **timeout**(D3X), **ttywait**(D3X)

Chapter 4

Data Structures (D4X)

Section D4X describes the data structures used by device drivers to share information between the driver and the kernel. The structures are presented on separate pages. All block and character driver data structures in the REAL/IX Operating System are identified with the (D4X) cross reference code.

Manual pages in this section contain the following headings:

DESCRIPTION	provides general information about the structure
STRUCTURE MEMBERS	lists all accessible structure members and defines the access permission for each. No attempt has been made to list these members in order; kernel code that you develop should not depend on specific locations of structure members.
SOURCE FILE	indicates the file name where the structure is defined
SEE ALSO	lists sources of additional information. The following abbreviations are used: *KPG* for the *Kernel Programming Guide* *DDG* for the *Driver Development Guide*

Overview of Kernel Data Structures

Data structures provide a means for passing information between the kernel and the driver routines. They are used to store process status information, to define I/O transfer methods, to define buffering schemes, and to store driver and device-specific information. There are basically three types of data structures:

- ❑ system data structures declared globally[1] for a driver
- ❑ driver-specific data structures declared globally for a driver
- ❑ internal data structures defined within a driver routine and used only by that routine

[1] A globally defined data structure is one that has been declared at the beginning of the driver code with a **#include** line or with an **extern** declaration.

Overview of Kernel Data Structures

The system data structures described in this section are structures that define common methods of passing information to and from the kernel and device drivers. Header files for these data structures are supplied with the delivered operating system in the */usr/include/sys* directory. Drivers declare the use of system data structures by adding the header file names with **#include** lines to the beginning of the driver code.

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 structures listed below are described in this section.

CAUTION

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.

- `areq` is the control block used for asynchronous I/O operations.
- `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).
- `cdevsw` contains system entry points for character driver routines.
- `cintr` contains information from the `cintrio`(4) structure that drivers may access.
- The following structures are used together for buffering character data:
  - `cblock` accesses character data array.
  - `ccblock` acts as a temporary buffer for unqueued characters.
  - `cfreelist` links a list of `cblocks`, headed by `chead`.
  - `clist` passes information between most tty drivers and the user program.
- `iobuf` is 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.

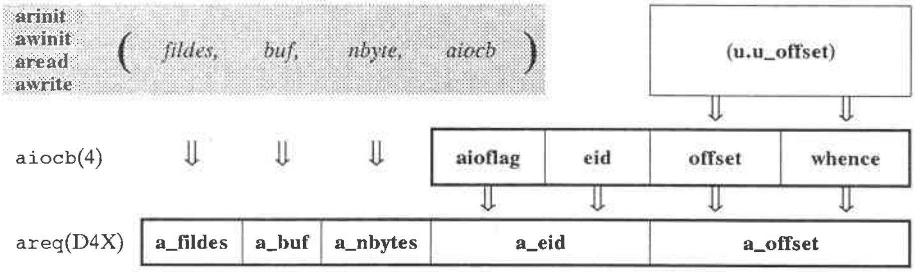
- sysinfo indicates the number of times a driver interrupt routine processes receive and transmit interrupts.

- tty controls character transfers between a TTY terminal driver and user data space.

- user defines the process and its current state.

DESCRIPTION

areq is the basic data structure used to control asynchronous I/O operations. It is populated from information in the aiocb(4) structure and the I/O request, as illustrated below.



Populating the areq Structure

Several areq structures can be allocated to one process simultaneously (the limit is determined by tunable parameters defining the number of asynchronous I/O operations per process and per system).

STRUCTURE MEMBERS

Type	Member	Description
char	*a_buf;	buffer pointer, in user virtual space
file_t	*a_fp;	associated file pointer
proc_t	*a_p;	pointer to process initiating the operation
uint	a_nbytes;	number of bytes to read or write
off_t	a_offset;	read/write character pointer
short	a_fildes	associated file descriptor
short	a_eid;	event id for posting; −1 if no event is to be posted
unchar	a_rw;	B_READ or B_WRITE operation
unchar	a_flags_1;	initialization status flags; may not be modified at interrupt level
unchar	a_flags_2;	status flags; may be modified at interrupt level
dev_t	a_dev	device on which to perform asynchronous I/O operation
int	a_dr_res[4];	available for driver-defined needs

All members of the areq structure (except a_dr_res[4]) are available to the driver for reading only; user-installed system calls should not access any members of **areq**. The members of the areq structure available to read by the driver are as follows:

a_buf points to the memory location of the buffer being used for this I/O operation. The buffer is in user virtual space; this area of the user's virtual memory is locked into physical memory before the driver is called. The driver must map the virtual memory to (possibly discontiguous) physical memory.

a_fp pointer to the file on which the I/O operation is being done.

a_p pointer to the process that initiated the I/O operation.

a_nbytes specifies the number of bytes to be transferred.

a_offset read/write character pointer. This member is populated based on the value of the **off_t** and **whence** members of the aiocb(4) structure, if any, and the current file offset.

If the file is a character special file, then the **a_offset** field is simply the byte offset implied by the **aread**(2) or **awrite**(2) system call. If the file is a regular, extent-based file, **a_offset** is set to the byte offset within the disk partition. For example, if an **aread** is to start from logical block 48 in a partition, **a_offset** will be assigned the value 48 * logical_block_size.

a_fildes *fildes* associated with this I/O operation.

a_eid event identifier to be posted when the I/O operation is complete. It is populated with the value of the **eid** member of the aiocb(4) structure if an event was specified; otherwise it is set to −1.

a_rw set to B_READ (read operation) or B_WRITE (write operation) to indicate the type of I/O requested.

a_flags_1 contains initialization status flags. When areq is initialized by **arinit** or **awinit**, both flags are set. Valid flags are:

ALINIT indicates that areq has been initialized by a previous call to **aread**(2), **awrite**(2), **arinit**(2), or **awinit**(2).

AIINIT indicates that `areq` has been initialized by **arinit**(2)/**awinit**(2)

a_flags_2 stores status information for the I/O operation. Valid flags are:

AINPROG indicates that an asynchronous I/O operation is in progress. It is set just before the `areq` is passed to the driver, and cleared when the driver calls the **comp_aio**(D3X) routine.

ACWAIT indicates that an asynchronous I/O operation is pending and a process is waiting. It is set when an operation is canceled because a file is closed, a process exited, or a process issued an **exec**(2). It is used to control a semaphore on which the process blocks awaiting completion of the operation, and is cleared when the driver calls the **comp_aio**(D3X) routine.

a_dev device on which to perform the asynchronous I/O operation. If the system call specifies a character special file, the device number is that of the raw device. If the file is a regular file, the device number is that of the block device.

a_dr_res[4] driver-settable if so defined by the application.

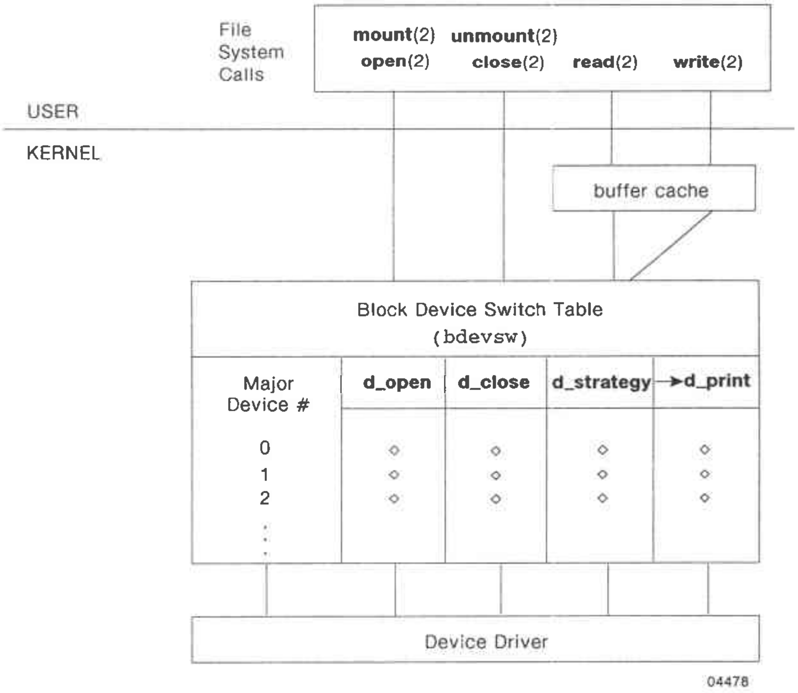
SOURCE FILE *os/aio.h*

SEE ALSO *KPG*, "Miscellaneous I/O Operations"
aio(D2X), **comp_aio**(D3X), **comp_cancel_aio**(D3X)
acancel(2), **aread**(2), **arinit**(2), **awrite**(2), **awinit**(2), **fcntl**(2), **aiocb**(4)

DESCRIPTION

The bdevsw (block device switch table) data structure provides kernel entry points into a driver. bdevsw is constructed when the system is initialized according to information provided to **sysgen**(1M). bdevsw is seldom accessed directly from the driver; if it is, all calls should be protected by the **drilock**(D3X) and **driunlock**(D3X) or **driinvoke**(D3X) kernel functions. The structure members section illustrates how the switch table appears in memory and in the */realix* file.

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 below.



bdevsw Structure

STRUCTURE MEMBERS

	Type	Member	Description
UNIX System V	`int`	`(*d_open)();`	Accesses driver **open**(D2X) routine
	`int`	`(*d_close)();`	Accesses driver **close**(D2X) routine
	`int`	`(*d_strategy)();`	Accesses driver **strategy**(D2X) routine
	`int`	`(*d_print)();`	Accesses driver **print**(D2X) routine
	`int`	`(*d_dump)();`	Accesses driver **dump**(D2X) routine
REAL/IX O/S only	`int`	`d_type`	Indicates how the driver is semaphored
	`int`	`d_cnt`	Number of minor devices supported
	`int`	`d_sems`	Pointer to driver semaphore structure

On the REAL/IX Operating System, three new fields have been added to `bdevsw` to configure the use of semaphores on a per-device basis. This enables you to port drivers developed for other UNIX operating systems to the REAL/IX Operating System without totally rewriting them for kernel semaphores.

The members of the `bdevsw` table used to semaphore the driver are as follows. These members should never be set or tested by the driver itself, but are populated according to information supplied to **sysgen**(1M) when the driver is installed.

- **d_type** indicates how the driver is semaphored. The valid values are:
  - 0 – driver code is semaphored and requires no additional preemption restrictions
  - 1 – driver runs on a specific CPU only and uses **spl*** functions to control interrupts
  - 2 – driver is protected from preemption with one semaphore per minor device
  - 3 – driver is protected from preemption by a single semaphore
- **d_cnt** is the number of minor devices supported; it is populated only if the driver is populated with one semaphore per minor device (**d_type** is 2)

- ❑ **d_sems** is a pointer to an array of **struct semdrivs**. The number of elements in the array is determined by **d_cnt**; the members of each element are defined on the semdrivs(D4X) manual page.

SOURCE FILE *sys/conf.h*

SEE ALSO **serv**(D2X), **drilock/undrilock**(D3X), semdrivs(D4X), user(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 needed to define a requested block I/O operation by specifying the device to be used, the direction of the data transfer, its size, the memory and device addresses, and other information. The kernel uses the information in the buffer header to organize and maintain the system buffer cache.

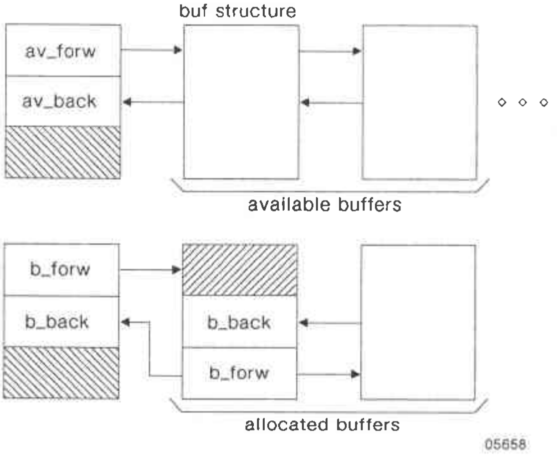
The buffer header pointer is the sole argument to a block driver **strategy**(D2X) routine. **strategy** typically uses the information in the buffer header to maintain an internal queue of I/O requests to be processed, and to return status information. Driver code uses pointers to refer to fields within the buffer header. For example, the following line uses the name *bp* as a pointer to the buffer header and specifies the **av_forw** member in that buffer header:

```
bp->av_forw
```

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. Do not depend on the size of the buf structure when writing a driver.

Buffer headers are also used by the system for paging user virtual memory to and from a swap device, and for unbuffered or physical I/O for block drivers. In this latter case, the buffer header is typically set up by the **physio**(D3X) routine and its subsidiary functions.

In the figure below, 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.



buf Structure

STRUCTURE MEMBERS

Type	Member	Description
int	b_flags;	Buffer status
struct buf	*b_forw;	Links the buffer into buffer cache hash queue
struct buf	*b_back;	Links the buffer into buffer cache hash queue
struct buf	*av_forw	Links buffer to free list or is available to driver
struct buf	*av_back	Links buffer to free list or is available to driver
dev_t	b_dev;	Major and minor device numbers
int	b_s1;	Available for driver use
int	b_s2;	
int	b_s3;	
sema_t	b_lock;	Semaphore for free buffer
sema_t	b_iodone;	Suspend semaphore indicating I/O done
unsigned	b_bcount;	Number of bytes to be transferred
caddr_t	b_addr;	Buffer's physical address
daddr_t	b_blkno;	Logical block number
char	b_error;	**u.u_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

Refer to the following table for structure member field use.

buf Structure Member Use

Member	Use	Member	Use
`b_flags`	driver settable; Do not clear	`b_bcount`	read only[c]
		`b_addr`	read only
`b_forw`	read only[a]	`b_blkno`	read only[c]
`b_back`	read only[a]	`b_error`	driver settable
`av_forw`	read only[b]	`b_resid`	driver settable
`av_back`	read only[b]	`b_start`	driver settable
`b_dev`	read only[c]	`b_proc`	read only[c]

[a] May be set by drivers that allocate the buffer themselves.
[b] May be set by drivers when buffer is not on the free list.
[c] May be set for raw I/O operations by drivers that allocate the buffer.

The members of the buffer header available to test or set by a driver are described below.

b_flags contains various flags that describe the buffer and any operation in progress. The member is a 32-bit integer. The most significant 16 bits are available for a driver to use with no restrictions; the least significant 16 bits contain flags that have meaning to the kernel.

Most of these flags are set by the kernel rather than the driver and care must be taken to preserve their values; B_ERROR can be set (but not cleared) by the driver, but the others have a number of subtle side effects if the driver sets them.

⚠ CAUTION

The driver must never clear the ***b_flags*** *member. If this member is cleared, unpredictable results can occur, including loss of disk sanity and the possible failure of other kernel processes.*

The valid flags are described below. Some of these flags are used only for the internal operation of the buffer cache, and of no concern to a driver. They are listed here for completeness, as they may be of use in understanding the state of the buffers in the buffer cache.

B_AGE signals to the **brelse**(D3X) function that the buffer should be placed at the head of the free queue when it is released, so it is reused before other buffers on the free queue

B_AIO indicates that the `buf` structure has been obtained with **getpbp**(D3X) for the purpose of controlling an asynchronous (non-blocking) I/O operation.

B_ASYNC set if operation is asynchronous. This implies that no user will be waiting on the **b_iodone** semaphore. This flag informs the **iodone**(D3X) function whether or not to issue a **vsema**(D3X) against **b_iodone** when the I/O transfer is complete. Drivers may make use of this information, such as in a request scheduling scheme that handles synchronous requests before asynchronous requests.

B_BUSY Historically, this flag was used to mark buffers that are in the "owned" state and not on the free queue. On the REAL/IX Operating System, this is handled with kernel semaphores, so this member is not used. However, drivers must preserve the value of this flag because it may be used in the debug kernel to provide an additional level of consistency checking.

A buffer can be in one of two states. If it is readily available for any process to use, it is on a free buffer queue and the **b_lock** semaphore has a value of 1, allowing the first process to do a **psema** operation to gain control of the buffer. Otherwise, the buffer is not on a free queue and the **b_lock** semaphore is set to 0 (indicating that the buffer is effectively "owned" by a process) or a negative number (indicating that it is owned and other processes are waiting for the buffer).

B_DELWRI set when a buffer contains data that is to be written out to a disk in a delayed write. The kernel will clear this flag before calling the driver to perform the actual write operation.

B_DONE Indicates the data transfer has completed. It is set by the **iodone**(D3X) function. The buffer cache code also uses this flag as an indicator that a buffer contains valid data.

B_ERROR set by the driver to indicate that an I/O transfer error has occurred. Error details can be given by setting the **b_error** member of the `buf` structure; if B_ERROR is set and **b_error** is not set, the kernel returns the default EIO error code.

If a process is waiting for the operation to complete, the **iowait**(D3X) function copies the error code from **b_error** to **u.u_error**, causing an error to be returned from the originating system call. When the buffer is eventually released, the B_ERROR flag causes the **brelse**(D3X) function to set the B_STALE flag. This occurs for both synchronous and asynchronous I/O operations.

B_FORMAT Used internally by certain drivers for some error logging operations.

B_OPEN Not used in `buf`, but is used in `iobuf`(D4X)

B_PHYS Set by kernel routines that use a buffer header for an I/O operation that does not use the system buffer cache, such as **physio**(D3X) and the routines that implement the virtual memory's demand paging scheme. This flag tells the driver that the transfer size given by the **b_bcount** member may be larger than the usual buffer cache transfer sizes.

B_READ Indicates data is to be read from the peripheral device into main memory

B_STALE Marks the buffer contents invalid; When the data in the buffer should not be used by a process looking in the cache, the kernel marks the buffer with this flag and places it at the head of the free queue for rapid reuse.

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 detected only as the inverse (NOT) of B_READ.

b_forw and **b_back**
Reserved for linking the buffer to a buffer cache hash queue.

av_forw and **av_back**
maintain the position of the buffer on the buffer cache freelist. When the buffer is not on the freelist, these members are available for driver use.

b_dev contains the external major and minor device numbers of the device accessed.

b_bcount specifies the amount of data (in bytes) to be transferred.

b_un.b_addr
normally, the kernel physical address of the data buffer controlled by the buffer header.[1] Data is read from the device to this starting address or is written to the device from this starting address. Occasionally, this member is used to hold a virtual address in user space, such as when a buffer is passed as a parameter to **disjointio**(D3X).

b_blkno identifies the logical block on the device (the device is defined by the minor device number) to be accessed. The block number is in terms of blocks with length BSIZE, which is 512 bytes on the REAL/IX Operating System. 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

[1] Note that, while all kernel addresses are technically virtual addresses, much of the kernel is mapped one-to-one to physical addresses and is called kernel physical memory.

conjunction with the B_ERROR flag in the **b_flags** member. Writing to this member overwrites any existing error code; to avoid this, check that `b_error == 0` (0 indicates no error) before writing the error code.

b_resid indicates the number of bytes not transferred because of an EOM or filemark or an no error condition.

b_start may be set up by the driver to hold the I/O operation start time. It can be used to measure device response time. Refer to the *Driver Development Guide*.

b_proc contains the process table entry address for the process requesting an unbuffered (direct) data transfer to a user data area.

paddr Macro

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 uses the **paddr** macro. The **paddr** macro is passed a pointer to a buffer header structure and returns the pointer to the buffer.

```
#include "sys/fs/s5param.h"

copy_the_data(bp)
struct buf *bp
{
     copyout(paddr(bp),u.u_base,bp->b_bcount);
}
```

SOURCE FILE

sys/buf.h

SEE ALSO

KPG, "Synchronized I/O Operations"
strategy(D2X), **physio**(D3X), **brelse**(D3X), **freepbp**(D3X), **getpbp**(D3X), **clrbuf**(D3X), **geteblk**(D3X), **getnblk**(D3X), `iobuf`(D4X)

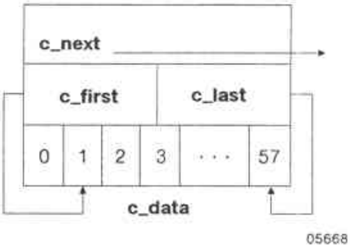
DESCRIPTION

Character data is stored in an array that is part of a cblock structure. cblock 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 58. Hence, a single cblock can contain up to 58 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 as illustrated in the figure below.

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.



cblock Structure

STRUCTURE MEMBERS

Type	Member	Description
struct cblock	*c_next	Pointer to the next cblock
char	c_first;	Index to the next **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

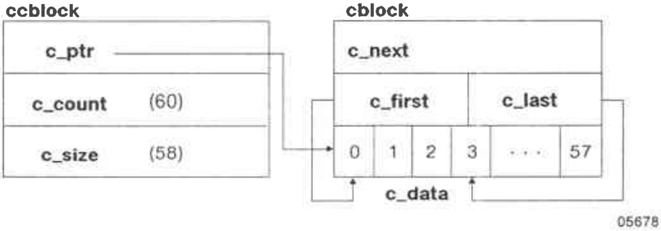
SOURCE FILE *sys/tty.h*

SEE ALSO *KPG*, "Drivers in the TTY Subsystem"
`ccblock`(D4X), `cfreelist`(D4X), `chead`(D4X), `clist`(D4X)

DESCRIPTION

The ccblock is the character control block used by the character I/O subsystem. 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. This is illustrated in the figure below.



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 accesses the **c_count** and **c_size** members of the cblock structure. **tp** is a pointer to the tty structure. Line 2 decrements **c_size** by **c_count**.

```
1   struct tty *tp
2   tp->t_tbuf.c_size = tp->t_tbuf.c_count;
```

STRUCTURE MEMBERS

Type	Member	Description
caddr_t	c_ptr;	Buffer address
ushort	c_count;	Character count
ushort	c_size;	Buffer size

SOURCE FILE

sys/tty.h

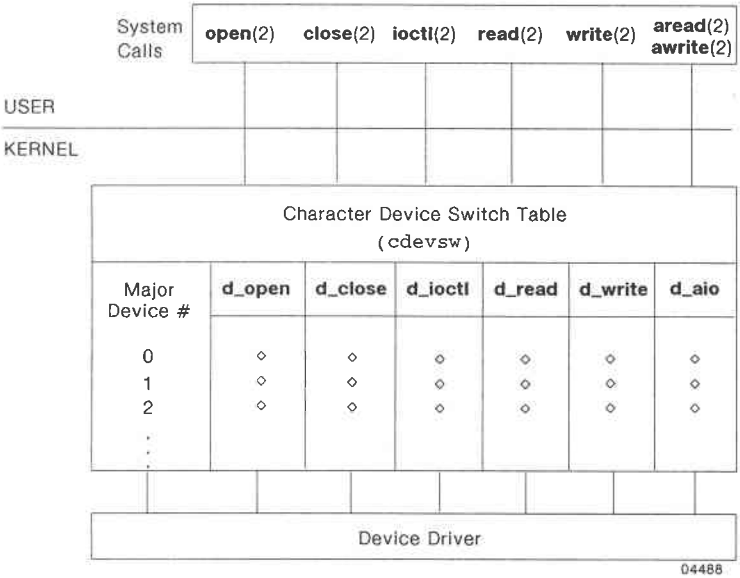
SEE ALSO

KPG, "Drivers in the TTY Subsystem"
cblock(D4X), cfreelist(D4X), chead(D4X), clist(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 as part of the configuration process from information given to **sysgen**(1M). cdevsw is seldom accessed directly from the driver; if it is, all calls should be protected by the **drilock/driunlock**(D3X) kernel functions. The structure members section illustrates how the switch table appears in memory and in the */realix* file.

The cdevsw table allows the kernel to map the names of devices to the device driver. The table includes pointers to functions used to implement user requests.



cdevsw Structure

STRUCTURE MEMBERS

Type		Member	Description
UNIX System V Entry Points	int	(*d_open)();	Accesses driver **open**(D2X) routine
	int	(*d_close)();	Accesses driver **close**(D2X) routine
	int	(*d_read)();	Accesses driver **read**(D2X) routine
	int	(*d_write)();	Accesses driver **write**(D2X) routine
	int	(*d_ioctl)();	Accesses driver **ioctl**(D2X) routine
Member for Async I/O	int	(*d_aio)();	Accesses driver **aio**(D2X) routine
Members for Polling	int	(*d_select)();	Accesses driver **select**(D2X) routine
	struct tty	*d_ttys;	Pointer to tty(D4X) structure
	struct streamtab	*d_str;	Pointer to stream table
Members for Semaphoring	int	d_type	Shows how the driver is semaphored
	int	d_cnt	Number of minor devices supported
	struct semdrivs	*d_sems	Pointer to driver semaphore structure
	short	d_dindx	Index into semdrivs(D4X) structure

Direct calls to cdevsw from within a driver should be protected with the **drilock**(D3X) and **driunlock**(D3X) or **driinvoke**(D3X) functions.

Member for Asynchronous I/O

The only entry point for asynchronous I/O is **aio**(D2X), which is accessed through the **d_aio** member of cdevsw. However, drivers that support asynchronous I/O must also support **ioctl**(D2X) commands from user processes issued with the GETAIOREQ command. This command returns information about asynchronous I/O, such as minimum and maximum transfer count. This information is available through the arwinfo structure in the *sys/fcntl.h* file.

Members for Polling

Device polling is implemented on the REAL/IX Operating System with the **select**(D2X) entry point plus pointers to two structures.

Members for Semaphoring Options

On the REAL/IX Operating System, four new fields have been added to cdevsw to configure the use of semaphores on a per-device basis. These compatibility modes enable you to port drivers developed for a similar operating system to the REAL/IX Operating System without rewriting them to use kernel semaphores.[1]

The members of the bdevsw table used to semaphore the driver are as follows. These members should never be set or tested by the driver itself, but are populated for the driver by **sysgen**(1M) when the kernel is built.

- **d_type** indicates how the driver is semaphored. The valid values are:
  - 0 – driver code is semaphored and requires no additional preemption restrictions
  - 1 – driver runs on a specific CPU only and uses **spl*** functions to control interrupts
  - 2 – driver is protected from preemption with one semaphore per minor device
  - 3 – driver is protected from preemption by a single semaphore
- **d_cnt** is the number of minor devices supported; it is populated only if the driver is populated with one semaphore per minor device (**d_type** is 2)
- **d_sems** is a pointer to an array of **struct semdrivs**. The number of elements in the array is determined by **d_cnt**; the members of each element are defined on the semdrivs(D4X) manual page.
- **d_dindx** is an index into the bdevsw(D4X) entry, used in drivers that support both block and character access.

SOURCE FILE

sys/conf.h

SEE ALSO

Section 2 in this manual
bdevsw(D4X), semdrivs(D4X)

[1] Not all compatibility modes are supported on all machines. Refer to the Release Notes shipped with your system.

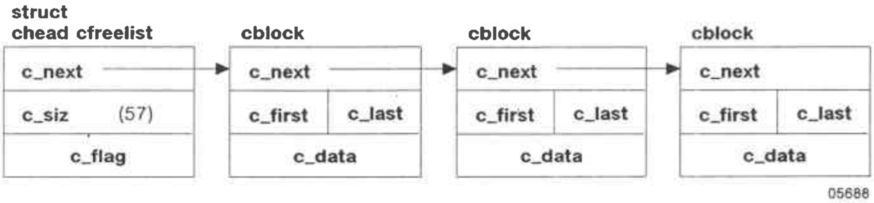
DESCRIPTION

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* description file.

The cfreelist is a singly linked list (**c_next**) of cblocks(D4X), as illustrated below. The **c_siz** variable in the clist head structure indicates the size of the cblock character buffer. Because 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.

CAUTION

The REAL/IX Operating System does not support the concept of blocking to wait for an available cblock structure. Rather, if a process tries to allocate a cblock when none is available, the system panics. To avoid this problem, always set the NCLIST tunable parameter to allocate more clists than can ever be used.



cfreelist Structure

STRUCTURE MEMBERS

Type	Member	Description
struct cblock	*c_next;	Singly linked list
int	c_siz;	Size of the cblock character buffer

SOURCE FILE

sys/tty.h

SEE ALSO

KPG, "Drivers in the TTY Subsystem"
cblock(D4X), ccblock(D4X), chead(D4X), clist(D4X)

DESCRIPTION

The cintr structure is the kernel connected interrupt data structure. It is populated with **cintrget**(D3X) from information in the cintrio(4) user-level data structure for connected interrupts, and released with **cintrelse**(D3X). The operating system moves information from cintr to cintrio as appropriate (usually after the **cintrnotify**(D3X) function is called).

STRUCTURE MEMBERS

Type	Member	Description
struct proc	*ci_procp;	pointer to connected process
lock_t	ci_lock;	spin lock
key_t	ci_key;	key; by convention, use the device number
int	ci_oneshot;	set if interrupt is in oneshot mode
int	ci_ack;	set if ci_oneshot is set and the interrupt has been acknowledged
int	*ci_pollptr;	pointer to user-mapped poll location
int	ci_cid;	current connected interrupt ID
sema_t	ci_sema	semaphore used with CINTR_SEMA method
struct cintrio	ci_ioctl	connected interrupt interface struct

All members of the cintr structure are readable by driver base-level and interrupt-level routines. Drivers should not set any field in the structure except with the IOCTL commands listed on the **cintrctl**(D3X) manual page.

SOURCE FILE

sys/cintrio.h

SEE ALSO

cintrctl(D3X), **cintrnotify**(D3X), **cintrelse**(D3X), cintr(D4X)
evctl(2), **evget**(2), **evrcv**(2), **evrcvl**(2), **evrel**(2), **cintrio**(4), **cintrio**(7)

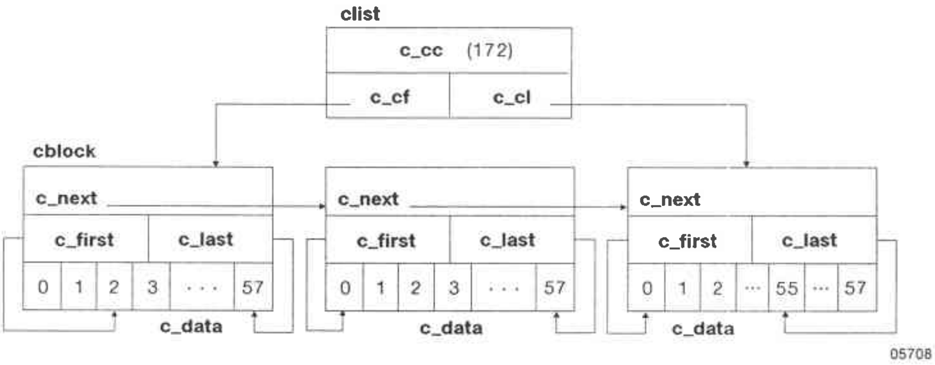
DESCRIPTION

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(D4X). It stores small quantities of data shared between a device and a user data area.

Typically, the terminal 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 58 characters (**c_data**) and maintain indexes that point to the first (**c_first**) and last (**c_last**) character in the buffer.

This clist structure in the figure below contains 172 bytes. This number is indicated by the value in **c_cc** member, as illustrated below.



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 *sys/tty.h*

SEE ALSO *KPG*, "Drivers in the TTY Subsystem"
cblock(D4X), ccblock(D4X), cfreelist(D4X), chead(D4X)

DESCRIPTION

Certain devices may operate with lists of transfer address/transfer count pairs that describe an I/O operation. The `djntio` structure defines an entry in such a list. Typically, an array of `djntio` structures is used to describe a collection of memory areas, with the last element of the array containing a zero count to mark the end of the list.

STRUCTURE MEMBERS

Type	Member	Description
`int`	`addr`	The start address of the area of memory described by the structure. Note that this would most naturally have a type "pointer to char" but an `int` type is used for reasons of compatibility with the porting base. When used with physical I/O devices, the address must be a physical address, not a virtual address. (Note that for most of kernel memory, the physical address will be identical to the virtual address.)
`int`	`count`	The number of bytes in the area of memory described by this structure.

SOURCE FILE

sys/disjointio.h

SEE ALSO

mbstrategy(D2X), **disjointio**(D3X), **djntget**(D3X), **djntfree**(D3X)

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. Most block device' driver **strategy**(D2X) routines require an internal queue to manage the device's outstanding I/O requests because the speed with which a typical block device can service requests is considerably slower than the speed with which requests can be made. **strategy** routines also need a structure to store specific 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 buf structures. These pointers can be used to create an internal request queue.

VME device controllers use the iobuf structure specifically. Each VME 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.

strategy routines that use the iobuf structure must declare the structure using the **extern** declaration in the driver's header file. The structure is a standard name constructed from the driver prefix in the form *prefix***tab**. For example, the iobuf structure for a driver with the prefix **doc_** would be:

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

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

STRUCTURE MEMBERS

Type	Member	Description
int	b_flags;	See buf(D4X)
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 registers to log on errors
paddr_t	io_addr;	Local bus address
struct iostat	*io_stp;	Unit I/O statistics
time_t	io_start;	Time that the I/O operation started
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 *sys/iobuf.h*

SEE ALSO buf(D4X)

DESCRIPTION

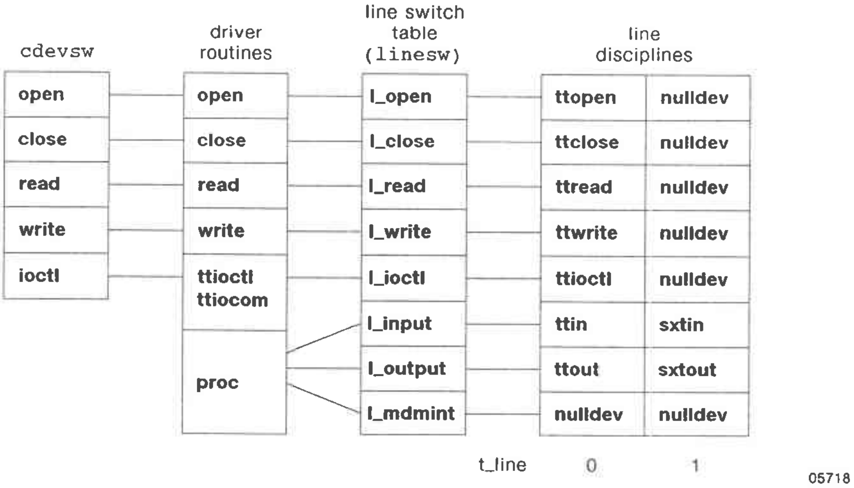
Line discipline is a term describing input/output character interpretation between the operating system and a terminal. It is the method by which characters are processed as they are sent and received from a terminal. The routines called by each attribute of a line discipline manipulate data in clists(D4X). 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 REAL/IX 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)), which cause previously entered information to be erased
- echoes 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.

The following figure illustrates how linesw translates a request for a line discipline function into a request for a **tt***(D3X) function.



linesw Structure

Valid line discipline values are 0 and 1. These values represent:

- ❑ Line discipline 0 is the TTY driver standard value.
- ❑ Line discipline 1 is used for *sxt* with **shl**(1), the shell layers 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 currently used, so it contains the address of 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 *sysgen/conf.c* program as shown in the following code segment.

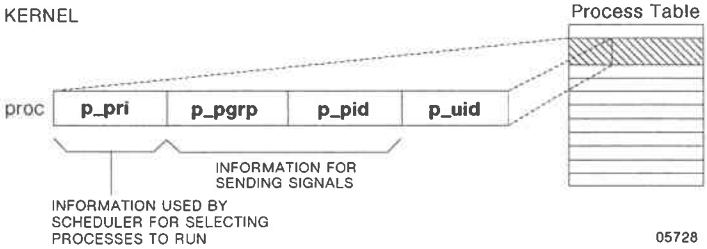
```
linesw[ ] = {
ttopen, ttclose, ttread, ttwrite, ttioctl, ttin, ttout, nulldev,
0
};
```

SOURCE FILE *sys/conf.h*

SEE ALSO Section 3 in this manual
KPG, "Drivers in the TTY Subsystem"

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 the figure below), used by memory management hardware and software to locate the code, data, and stack information of the process. It also contains information used by the scheduler in selecting processes to run.


KERNEL
Process Table
proc
p_pri
p_pgrp
p_pid
p_uid
INFORMATION FOR SENDING SIGNALS
INFORMATION USED BY SCHEDULER FOR SELECTING PROCESSES TO RUN
05728

proc Structure

The *process table* is an array of proc data structures. Each process known to the kernel is described by one, arbitrarily picked, array entry in this table. The entry contains everything the kernel needs to control that process, or pointers to where such information is stored. For example, the *process id* is stored in that process's proc data structure; the memory management unit (MMU) maps for that process are stored elsewhere, with a pointer to their location kept in the proc structure. Thus, the proc structure may be considered to be the root of all information the kernel has about a process.

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

The proc structure can be viewed using the **crash proc** command.

STRUCTURE MEMBERS

The following members of the proc structure may be read by a driver or system call. proc structures are subject to change from one software release to another; the members listed here are not expected to change in future releases.

CAUTION

Drivers and user-installed system calls should never modify the proc *structure directly.*

Type	Member	Description
uint	p_flag;	Flags
lock_t	p_lock	Must be locked before calling **psignalval**(D3X)
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, used to send signals to a group of processes
short	p_pid;	Process identification number, used to send a signal to a specific process
short	p_ppid;	Process identification number of parent process
ushort	p_sgid;	Effective group id (set by **exec**(2))
int	p_sig;	Signals pending to this process
uint	p_size;	Size, in pages, of the process swappable image
short	p_slp_cnt;	Pointer to counter that can be used to track **vsema**(D3X) calls associated with interruptible **psema**(D3X) calls
ushort	p_suid;	Effective user id (set by **exec**(2))
char	p_stat;	The status of the process, used by the scheduler
ushort	p_uid;	Process user id

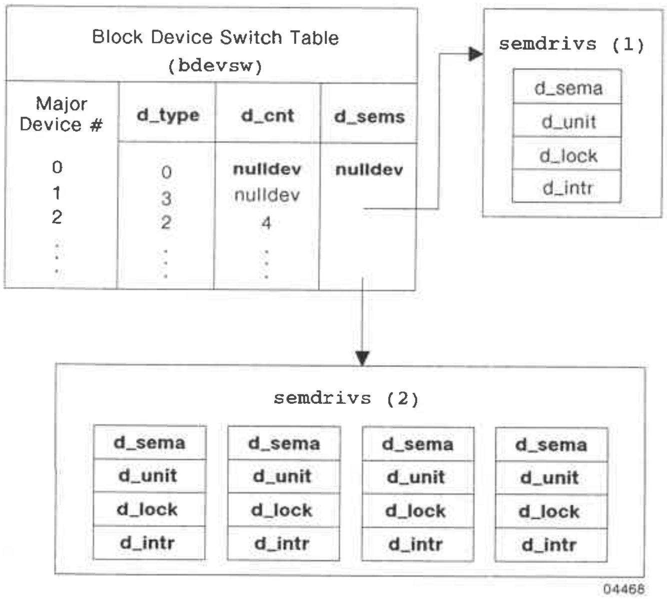
SOURCE FILE

sys/proc.h

DESCRIPTION

The semdrivs data structure is used with drivers that are installed under either major or minor number semaphoring compatibility modes. The **d_sems** member of the switch table entry points to an array or semdrivs structure; the number of semdrivs structures is indicated by the **d_cnt** member of the switch table.

The figure below illustrates how the switch table points to a semdrivs array; the example is for bdevsw(D4X), but would be the same for cdevsw(D4X).



Block Device Switch Table (bdevsw)

Major Device #	d_type	d_cnt	d_sems
0	0	**nulldev**	**nulldev**
1	3	nulldev	
2	2	4	
⋮	⋮	⋮	

Accessing semdrivs from a Switch Table

In the figure, Major Device #1 is semaphored on the major number (d_type=3), so semdrivs is an array of one element. Major Device #2 is semaphored on the minor number (d_type=2), so semdrivs is an array of **d_cnt** members, where **d_cnt** is a member of the switch table structure, indicating the number of minor devices supported (in this example, 4).

STRUCTURE MEMBERS

Type	Member	Description
sema_t	d_sema	address of the driver semaphore
int	d_unit	bit map of the units needing service
int	d_stype	identifies type of semaphoring for **sleep**(D3X) and **serv**(D2X)
lock_t	d_lock	spin lock to protect **d_unit**
int	(*d_intr)();	pointer to the device interrupt routine
int	d_mult	used to associate bit number with minor device

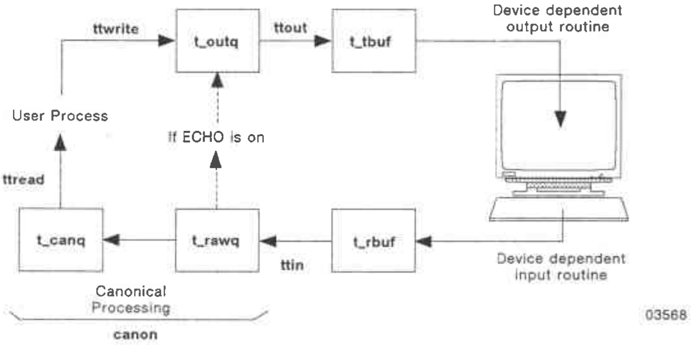
SOURCE FILE *sys/conf.h*

SEE ALSO *DDG*, "Porting Drivers"
bdevsw(D4X), cdevsw(D4X)

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, as illustrated below.



Using the tty Structure

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
- echoes 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 form **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 form 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.

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 ccblock	t_tbuf;	Device transmit buffer
struct ccblock	t_rbuf;	Device receive buffer
int	t_rsel;	Select attempted on this device for read
int	t_wsel;	Select attempted on this device for write
int	(*t_proc)();	proc routine address
tcflag_t	t_iflag;	Input mode
tcflag_t	t_oflag;	Output mode
tcflag_t	t_cflag;	Control mode
tcflag_t	t_lflag;	Local mode
ulong	t_state;	Device and driver internal state
short	t_pgrp;	Process group name
char	t_line;	Line discipline type
char	t_delct;	Number of delimiters
char	t_term;	Terminal type
char	t_tmflag;	Terminal flag
char	t_col;	Current column
char	t_row;	Current row
char	t_vrow;	Variable row
char	t_lrow;	Last physical row
char	t_hqcnt;	Number of high queue packets on **t_outq**
char	t_dstat	Used by terminal handlers and line disciplines
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 device's transmit buffer

t_rbuf 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

RCOLL indicates there was a collision in read **select**

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 blocked 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.

WCOLL indicates there was a collision in **write** select

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]
array holding the control characters specified in the **c_cc** member of `termio`

The `tty` structure contains other members used to implement CPU affinity for a TTY device; these members are never accessed directly by the driver.

A character device driver using the TTY subsystem must declare an instance of the `tty` structure for each subdevice under its control.

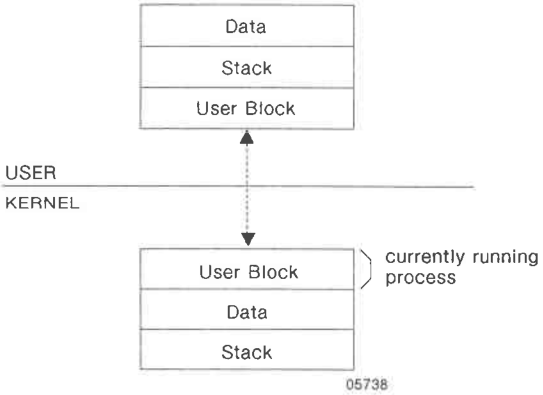
SOURCE FILE *sys/tty.h*

SEE ALSO *KPG*, "Drivers in the TTY Subsystem"
`linesw`(D4X)

DESCRIPTION

The user structure[1] defines the fields included in the user block for each process. It may be thought of as an extension to the proc(D4X) structure, which holds control information about a process that can be rolled out whenever the process itself is rolled out. User blocks are created dynamically for each newly created process. The process user block contains information such as where the data is coming from, its size, and how much needs to be moved. Character driver **read**(D2X) and **write**(D2X) routines may use these fields to read information they need about the status of an I/O request, and to write the I/O request's final status.

When a process begins to execute in the CPU, the user block for the process is placed at a fixed address in the kernel; this location is called the u_area. Only one user process can run on a given CPU at one time. This means that the user block in the CPU is always the block for the currently running process. A new process that has a higher priority than the process currently running may cause that process to be preempted, in which case a new user block is swapped in for the higher priority process. For this reason, **strategy**(D2X) and interrupt-level routines (**intr**(D2X) and **serv**(D2X)) must not access the user structure. These routines operate independently of the currently running process and could inadvertently alter the fields of a user block for a process not associated with them.



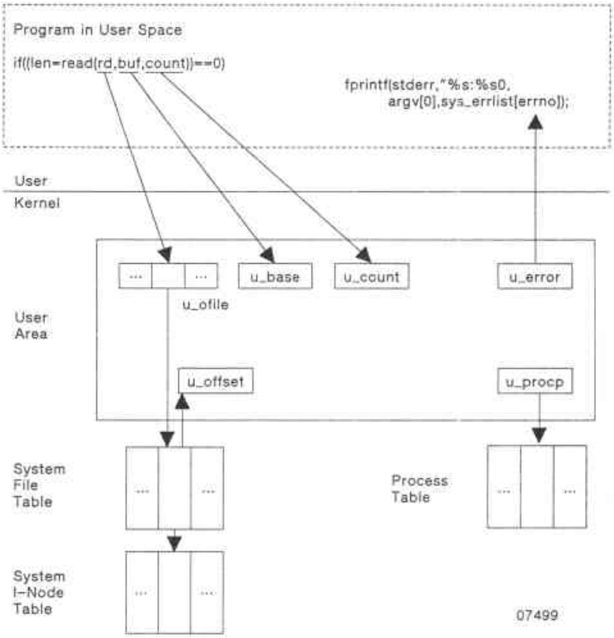
user Structure

[1] The user structure is also commonly called the **u** structure or **u** block, and sometimes is referred to as the user area (u_area). User area should not be confused with user address space, which refers to the part of memory in which a user-level process executes.

Most fields defined in the *user.h* header file are pertinent only to character I/O **read** and **write** routines. **init**, **open**, **close**, and **ioctl** routines can also access the `user` structure, although the **u.u_base** and **u.u_count** fields that define the size and location of the data transfer are not meaningful to these routines. Block I/O requests are handled through the system buffer cache defined by the `buf`(D4X) structure.

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, as illustrated below.

The `user` structure is populated from a system call, as illustrated below.



```
if((len=read(rd,buf,count))==0)
```

```
fprintf(stderr,"%s:%s0,
        argv[0],sys_errlist[errno]);
```

Populating the user structure from a system call

All members of the user structure shown in this diagram are explained on the pages that follow in this section, except **u.u_ofile**, which is the first in an array of pointers to file table entries for open files.

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 a currently running process. Because the user structure is basic to the kernel, it is subject to change from one software release to another.

STRUCTURE MEMBERS

Type	Member	Description
int	*u_ap;	Pointer to argument list (**uap** macro)
int	*u_ar0[0];	Data to return to user process (**rval1** macro)
int	*u_ar0[1];	Data to return to user process (**rval2** macro)
int	u_arg[ ];	Arguments to current system call
caddr_t	u_base;	I/O base address
unsigned	u_count;	Bytes remaining for I/O
int	u_error;	Return error code
short	u_fmode;	File mode for I/O
gid_t[a]	u_gid;	Effective group ID
off_t	u_offset;	Offset into file for I/O
int	u_preempt;	Flags to disable preemption
struct proc	*u_procp;	proc structure pointer
gid_t[a]	u_rgid;	Real group ID
unsigned char	u_rt;	Checks realtime privileges
uid_t[a]	u_ruid;	Real user ID
char	u_segflg;	User or kernel I/O flag
char	u_nshmseg;	Number of shared memory segments attached
short	*u_ttyp;	Pointer to pgrp in tty(D4X) structure
uid_t[a]	u_uid;	Effective user ID

[a] ushort in machines with MVME680x0 MPU or Intel 80x86 CPU.

These members of the user structure are described as follows:

u_ap points to the argument list for the current process; is usually accessed with the **uap** macro, which should be defined in the code as follows:

```
*uap = (struct a *) u.u_ap;
```

u_ar0[0] used to return information from a system call; accessed with the **rval1** macro

u_ar0[1] used like **u_ar0[0]** when a second piece of information must be returned from a system call; accessed with the **rval2** macro

u_arg[ ] arguments passed from the current system call

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 (refer to *errno.h*) to the kernel; the error code is then passed on to the user. This field is set by a driver to indicate an error condition. See **intro**(2) for a description of available error codes for setting error codes. Also refer to **copyin**(D3X) for an example of the **u.u_error** member.

u_fmode copy of the **f_flag** member of the file structure (defined in *sys/file.h*). The flag propagates the modes set in the **open**(2) request.

u_offset specifies the offset into the file from which or to which data is being transferred

u_preempt flags to disable kernel preemption

u_procp address of the proc(D4X) structure associated with this user structure

u_rt defines whether the process is executing with realtime privileges; is set and checked with the **rtuser** macro

u_ruid and **u_rgid**
identifies the real user and group IDs

u_rval1 and **u_rval2**
point to registers that store values to be returned to the user

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 0 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_nshmseg
number of shared memory segments attached to this process

u_ttyp
address of the tty(D4X) 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.

The following table lists `user` structure members that do not vary between UNIX System releases and that can be set or read.

Access Rules for user Structure

Member	Use	
	Drivers	System Calls
u_ap	do not access	read with **uap** macro
u_ar	do not access	read only
u_ar0[0]	do not access	set with **rval1** macro
u_ar0[1]	do not access	set with **rval2** macro
u_arg[6]	do not access	read only
u_base	driver settable	read only
u_count	driver settable	read only
u_error	driver settable; do not clear	settable; do not clear
u_fmode	do not access	
u_gid	read only	read only
u_offset	driver settable	
u_preempt	do not access	read only
u_procp	read only	read only
u_qsav	read only	do not access
u_rgid	read only	read only
u_ruid	read only	read only
u_segflg	driver settable	
u_nshmseg	do not access	read only
u_syscall	do not access	read only
u_ttyp	driver settable	read only
u_uid	read only	read only

SOURCE FILE *sys/user.h*

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

MODCOMP, founded in 1970,
is a worldwide supplier of
real-time systems, products,
and services. MODCOMP is an
AEG company.

Corporate Headquarters:
Modular Computer Systems, Inc.
1650 West McNab Road
P.O. Box 6099
Ft. Lauderdale, FL 33309-1088
Tel: (305) 974-1380
Twx: 510-956-9414

International Headquarters:
Modular Computer Services, Inc.
The Business Centre
Molly Millars Lane
Wokingham, Berkshire
RG11 2JQ, UK
Tel: 0734-786808
Fax: 0734-776399

Americas Headquarters:
Modular Computer Systems, Inc.
1650 West McNab Road
P.O. Box 6099
Ft. Lauderdale, FL 33309-1088
Tel: (305) 974-1380
Twx: 510-956-9414

MODCOMP sales and service
offices are located throughout
the world.

Printed in U.S.A. • "The technical contents of this document, while accurate as of the date of publication, are subject to change without notice."