SINIX Open Desktop

Development System Device Driver Writer's Guide

Edition April 1990 (SINIX Open Desktop V1.0)

Order-No. U5753-J-Z95-1-7600
Printed in the Federal Republic of Germany
7760 AG 2900.5 (9700)

Published by Bereich
Daten- und Informationstechnik
Postfach 83 09 51, D-8000 München 83

Siemens Aktiengesellschaft

SCO UNIX® System V/386

Development System

Device Driver Writer's Guide

The Santa Cruz Operation, Inc.

SCO Document Number: 6-26-89-6.0/3.2.0

Contents

1 **Introduction**

About This Book 1-1
The Driver Development Package 1-5

2 **Writing a Device Driver**

Introduction 2-1
Kernel Environment 2-13
Memory Management 2-19
Direct Memory Access (DMA) 2-28
Kernel Routine Summary 2-36
Driver Development Considerations 2-39
Sample Drivers 2-43

3 **Block Device Drivers**

Introduction to Block Devices 3-1
Example Block Driver Code 3-14

4 **Character Device Drivers**

Introduction to Character Devices 4-1
Example Character Driver Code 4-14
Code Fragments from a Line Printer Driver 4-15
Code Fragments From a Terminal Driver 4-21

5 **Video Adapter Device Drivers**

Writing a Video Adapter Driver 5-1
Video Driver Structures 5-8
Adapter Driver Routines 5-15

6 **Compiling and Linking Drivers**

Compiling, Configuring, and Linking Drivers 6-1
Driver Debugging 6-6
Notes On Preparing a Driver for Binary Distribution 6-11

7 **Writing a SCSI Driver**

Introduction 7-1
Driver Overview 7-5
Request Block 7-8
Writing a SCSI Device Driver 7-11
Installing a SCSI Device or Host Adapter Driver 7-18
Host Adapter Driver 7-20

8 **Line Disciplines**

Introduction 8-1
tty Structure 8-7
The xxproc Routine 8-12
ttiocom Routine 8-16

9 **STREAMS**

Overview 9-1
STREAMS Driver Overview 9-4
STREAMS Loop Back Driver 9-8
STREAMS Test Program 9-16

A **Migrating XENIX Drivers to the System V Operating System**

Introduction A-1
Binaries A-2
Change in the Device Number A-3
Changes in Structure Definitions A-4
Changes in Kernel Routines A-8
Obsolete Kernel Routines A-12
General Notes A-14
Converting IDDs A-17

B **Sample Block Driver**

Overview B-1
Block Driver Header File B-2
Block Driver B-4

C **Section (K) Manual Pages**

Manual Page Overview C-1

Chapter 1

Introduction

About This Book 1-1

- What is a Device Driver? 1-1
- Scope 1-1
- Audience 1-2
- Contents Overview 1-2
- Naming Conventions 1-4
  - Guide Conventions 1-4
  - Section (K) Manual Pages Conventions 1-4
  - Header Files Conventions 1-4

The Driver Development Package 1-5

About This Book

This guide describes how to write or maintain a device driver. Included are descriptions of the various driver categories that you are likely to see in the device driver spectrum. In addition, the section (K) manual pages at the end of this guide provide a set of routines you can use to write a driver.

This guide provides you with the instructions for creating and installing a device driver. Information is also given about the routines that you write and those provided for you. The routines provided in System V are all contained in your binary copy of System V; no further installation is required to create and install a driver using System V.

NOTE: If you are porting drivers from XENIX, read Appendix A, "Migrating XENIX Drivers to the System V Operating System" for more information.

What is a Device Driver?

For each peripheral device (such as a terminal) there must be a *device driver* to provide the software interface between the device and the operating system. A device driver is:

- A set of routines that communicates with a hardware device and provides a uniform interface to the operating system kernel.
- A manager of data flow and control between a user program and a peripheral device.
- A user-defined section of the kernel that allows a program or a peripheral device to appear as a */dev* device to the rest of the computer's software.

Scope

This guide provides information about developing or maintaining a driver using only the SCO System V/386 operating system software; applicability to other vendor's System V operating system software is outside the scope of this guide.

Audience

This guide is meant as a reference for the experienced C Language developer who wishes to use previously unsupported hardware. Writing a device driver is complex. You must have a technical reference for both the device you wish to support and for the computer containing the driver and device.

Contents Overview

The contents of this guide are as follows:

Chapter 1 — Introduction
A description of this book.

Chapter 2 — Writing a Device Driver
A description of driver development information. This chapter includes both conceptual and practical information about writing or maintaining a device driver. The chapter discusses driver concepts, a description of the kernel environment, how to manage memory, how to use direct memory access, considerations for driver development, a summary of the kernel routines in section (K), and a description of the sample drivers provided in the software that accompanies this guide.

Chapter 3 — Block Device Drivers
A description of the routines that you write when creating or maintaining a block device driver. A block device driver handles interaction with disk and tape devices, for example, a floppy disk drive. Included in this chapter are fragments from a sample disk driver.

Chapter 4 — Character Device Drivers
A description of the routines that you write when creating or maintaining a character device driver. A character driver handles interaction with devices such as terminals and printers. Included in this chapter are fragments from two character drivers, a line printer driver and a terminal driver.

Chapter 5 — Video Adapter Device Drivers
A description of the routines that you write when creating or maintaining a video adapter driver. A video driver works with monochrome (mono), CGA, EGA, and VGA controllers on a computer to provide the functionality necessary to display intricate graphics and/or high-speed displays.

Chapter 6 — Compiling and Linking Drivers
A description of how to make your driver source code part of the System V kernel. Described are the methods for compiling a driver and how to use the Link Kit to create a new kernel. A section on debugging shows how to get a driver to run correctly, and what to do if it does not.

Chapter 7 — Writing a SCSI Driver
A description of the small computer systems interface (SCSI) adapter code supplied with your system and how to write a SCSI driver to interact with this code.

Chapter 8 — Line Disciplines
A description of the concepts and procedures for writing, maintaining, and installing a line discipline.

Chapter 9 — STREAMS
A description of STREAMS followed by a complete working driver for a loop back device.

Appendix A — Migrating XENIX Drivers to System V
A description of how to port XENIX drivers to System V.

Appendix B — Sample Block Driver
A sample working floppy disk driver and its header file.

Appendix C — Section (K) Manual Pages
The manual pages for the section (K) kernel routines.

Naming Conventions

Naming conventions for the information shown in this book is described in the following tables.

Guide Conventions

The following conventions are used in this guide to show computer names:

structures, *structure members,* *filenames, pathnames*	*Italics*
routine names, **section (K) routine names**	**Bold**
`code examples`	`Courier`

Section (K) Manual Pages Conventions

In the manual pages in section (K), the following conventions are used:

structures, **structure members,** **filenames, pathnames**	**Bold**
routine names, *section (K) routine names*	*Italics*
`code examples`	`Courier`

The two exceptions to these guidelines for section (K) are that the **Syntax** section is all in **bold**, and the *See Also* section is all in regular type.

Header Files Conventions

In System V, header files are located in the */usr/include/sys* directory. In this guide, references to header file pathnames are abbreviated to *sys/filename.h*. For example, video adapter driver information is located in the *sys/vid.h* header file.

The Driver Development Package

The driver development package supplied with your software includes these parts:

- *Device Drivers Writer's Guide* — This guide provides information about developing and installing a driver, and includes the section (K) manual pages at the end of the book. All section (K) manual pages are accessible with the **man**(C) user command when the K section is specified, for example, use `man K Intro` to list the introduction section (K) manual page.

- *Driver Development Sample Drivers* — A series of working drivers provided with the SCO System V/386 development system software. Use these drivers and the installation script provided in the software to examine a driver, install it, and test its use on your system. These drivers contain examples of the routines and concepts described in this guide. The sample drivers are described in more detail in Chapter 2 in the "Sample Drivers" section.

- *SCO System V/386 Documentation and Online Manual Pages* — The operating system and development system manual pages and reference manuals that are provided with your system documentation. Of special interest to driver developers are the following manual pages that can be listed with the **man**(C) user command, and relevant manuals:

  Driver Development

  - Line Discipline Drivers — **termio**(M)
  - Video Drivers — **multiscreen**(M), **mvdevice**(F), **screen**(HW)
  - STREAMS Drivers — **clone**(STR), **streamio**(STR), and the *STREAMS System* manual.

  Installation and Debugging

  - Installation — **cc**(CP), **configure**(ADM), **custom**(ADM), **fixperms**(ADM), **masm**(CP), **mdevice**(M), **mtune**(F), **sdevice**(F), **stune**(F)
  - Debugging — **crash**(ADM), **ps**(C)

Chapter 2

Writing a Device Driver

Introduction 2-1
- Driver Design 2-2
- Routines in a Device Driver 2-3
- Block and Character Devices 2-7
- Differences Between a Driver and an Application Program 2-9
- Special Device Files 2-11

Kernel Environment 2-13
- What is an Interrupt? 2-13
- Modes of Operation 2-13
- Context Switching 2-14
- The System-Mode Stack 2-14
- Task-Time Processing 2-16
- Interrupt-Time Processing 2-16
- Interrupt Routine Guidelines 2-18

Memory Management 2-19
- Physical Memory Allocation 2-19
- Allocating Dynamic Memory 2-20
- Memory-Mapped I/O Allocation 2-24
- Allocating Memory to Share With User Processes 2-25

Direct Memory Access (DMA) 2-28
- Allocating Memory for DMA Physical I/O 2-28
- DMA Routines Provided In the Kernel 2-30
- DMA Transfer 2-31
  - Managed DMA (Non-Queued) 2-31
  - Queued DMA Transfer 2-33

Kernel Routine Summary 2-36

Driver Development Considerations 2-39
- What To Do and What Not To Do 2-39
- How Data is Passed To a Driver 2-41
- Sharing Interrupt Vectors 2-42

Sample Drivers 2-43

Introduction

This chapter contains background information about writing a device driver. This information is useful for writing a driver for the first time or maintaining an existing driver. Both conceptual and practical information are presented including descriptions of drivers and a high-level description of the kernel routines used when coding a driver.

The topics described in this chapter are:

- **Introduction** — Concepts of driver development
- **Kernel Environment** — Aspects of the kernel that support driver development
- **Memory Management** — Concepts and practical information about managing memory in a driver
- **Direct Memory Access (DMA)** — How to use DMA in a device driver
- **Kernel Routines Summary** — Routines that are used to write or maintain a device driver; an introduction to the section (K) routines described in detail at the end of this book
- **Driver Development Considerations** — Ideas to consider when developing a driver, information about parameters are passed to a driver, and a description of how to share interrupts in a driver
- **Sample Drivers** — Descriptions of the sample drivers provided with your development system software

If you are an experienced driver writer, you may wish to skip the conceptual information at the front, but the rest of the chapter should be read for applicability to your driver.

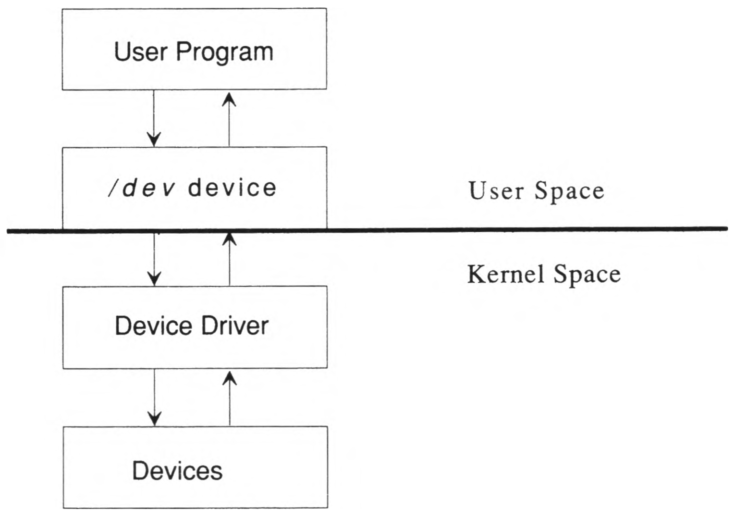
NOTE: If you are porting drivers from XENIX, read Appendix A, "Migrating XENIX Drivers to the System V Operating System" for more information.

Driver Design

The implications of a device driver are numerous, some are as follows:

- When writing a device driver, you are given an opportunity to design a section of the kernel to fit your needs.
- Because you are adding to the existing supported kernel, ensure that the code you write does not corrupt existing applications or alter the way the kernel behaves.
- Only use the functionality described in this guide; this ensures that your driver is portable and one that can be supported readily.
- Always backup existing software before implementing a device driver.
- Provide tunable parameters to give end users access to the capabilities and features of your driver. In addition, provide I/O control capability and document parameters and controls both in the code and in the header file.
- Thoroughly document a device driver including its purpose and operation.
- Thoroughly test your driver to ensure that it works correctly.
- Create an installation package so that your driver can be installed easily. Ensure that the installation process is well documented and tested.

The following diagram shows the relationship between a driver and the operating system:



Routines in a Device Driver

A driver consists of *kernel routines* and *driver routines*. *Kernel routines* are provided in the operating system and are described in the section (K) manual pages that are in this guide.

Driver routines are routines that you write. The names of these routines have a two-part format: a two-, three-, or four-character prefix that is unique to your driver, and a routine name suffix. In this guide, this format is expressed as **xx***routine-name*, for example, the driver read routine is called **xxread**. "**xx**" is a generic prefix name. Your driver's prefix should be unique and relevant to the device you are accessing, such as **ramd** for a RAM disk driver.

Driver routines are described in many of the chapters in this guide by the type of device that the driver is servicing — block, character, video, or STREAMS.

A device driver is written for either block or character devices:

— Data conveyed between a block device and the operating system is buffered in uniform blocks of bytes.

— Data conveyed from a character device is in arbitrary amounts and may or may not be buffered.

Examples of block devices are disk or tape drives. Examples of character devices are terminals, printers, and network cards.

The following table describes the routines that you need to write when creating a driver for a block, character, or video driver (however, not all driver routines are required by all drivers):

Block Device Driver Routines

Routine	Purpose
xxopen	Start access to a block device
xxclose	End access to a block device
xxstrategy	Perform buffered I/O with device
xxprint	Display message on the console
xxread	Perform raw reads on a block device
xxwrite	Perform raw writes to a block device
xxioctl	Perform I/O control commands
xxbreakup	Size I/O request into usable chunks
xxd_proc	Perform queued DMA request
xxinit	Initialize the device when the system boots
xxintr	Handle interrupt from a block device
xxstart	Access device-specific I/O ports

The first set of routines in this table are entry points required by the kernel. The second set is routines used in physical I/O. The third set is optional driver-specific routines. Block device driver routines are described in Chapter 3, "Block Device Drivers." The **xxd_proc** routine is described the "Direct Memory Access (DMA)" section in this chapter.

Character Device Driver Routines

Routine	Purpose
xxopen	Start access to a character device
xxclose	End access to a character device
xxread	Transfer data from internal buffers to user space
xxwrite	Transfer data from user space to internal buffers
xxioctl	Perform I/O control commands
xxinit	Initialize the device when it is booted
xxhalt	Executed when the computer is shut down
xxintr	Executed when an interrupt occurs
xxstart	Interact directly with the device
xxpoll	Executed on each clock tick
xxproc	Perform device-dependent line discipline I/O

The first set of routines is the entry point routines required in a character driver. The other set is optional driver-specific routines. Character device driver routines are described in Chapter 4, "Character Device Drivers." The **xxproc** routine is described in Chapter 8, "Line Disciplines."

Video Adapter Device Driver Routines

Routine	Purpose
xxadapctl	Support video adapter functionality
xxclear	Clear any portion of the screen
xxcmos	Determine if a video card is the primary video card
xxcopy	Copy data between screen areas
xxinit	Called when the system is booted
xxinitscreen	Called each time a multiscreen is created
xxioctl	Perform I/O control commands
xxpchar	Write data beginning at the current cursor position
xxscroll	Scroll the screen up or down
xxscurs	Update hardware cursor position
xxsgr	Support ANSI terminal functionality

All routines shown in this table are entry points for a video adapter device driver. These routines are described in Chapter 5, "Video Adapter Device Drivers."

SCSI Device Drivers

Routine	Purpose
xxopen	Start access to SCSI devices
xxclose	End access to SCSI devices
xxstrategy	Manage buffers to be sent or received from devices
xxprint	Display error messages
xxread	Build read request blocks
xxwrite	Build write request blocks
xxioctl	Control adapter-specific features
xxstart	Send request blocks to the device and process any returned error information

The first set of routines are the entry point routines that are required for a device driver. The second set of routines perform physical I/O. The last set are specialized routines. SCSI device driver routines are described in Chapter 7, "Writing a SCSI Driver."

SCSI Host Adapter Drivers

Routine	Purpose
xxopen	Start access to a SCSI bus
xxclose	End access to a SCSI bus
xxioctl	Control adapter-specific features
xxintr	Handle SCSI bus interrupts
xxinit	Initialize SCSI access
xx_entry	Common entry point to SCSI device drivers for communication with a host adapter driver.

The first set of routines are the entry point routines that are required for a host adapter driver. Other entry point routines normally used in a block driver are stubbed out so that NULL is returned when called. The second set are specialized routines. Host adapter driver routines are described in Chapter 7, "Writing a SCSI Driver."

STREAMS Drivers

Routine	Purpose
xxopen	Open a STREAMS device or module
xxclose	Discontinue device or module access
xxput	Put messages on a queue
xxsrv	Service messages on a queue

The first set of routines replace the entry point routines for the STREAMS character driver. The second set of routines are used when accessing messages on a STREAMS read or write queue. STREAMS drivers are described in Chapter 9, "STREAMS."

Block and Character Devices

In general, any device with a randomly addressable set of fixed-size records is a block device; any other type of device is a character device. For example, disk drives are block devices, while terminals and line printers are character devices. The operating system presents a uniform interface to user programs by coding device dependent issues inside the device drivers. User processes can access devices just as they would a regular file. The kernel and the associated device driver perform the necessary transformations to change a user request, such as **read**(S), to an I/O request for the device. Thus, character and block devices look alike to the user program.

Character Device Drivers

Character-device drivers can communicate directly with the user program. Driver access begins when a user program requests a data transfer of some number of bytes between a section of its memory and a specific device. The operating system transfers control to the appropriate device driver. The user program supplies the parameters for the request to the device driver, which in turn performs the work. Thus, the operating system has minimal involvement in the request; the data transfer is a private transaction between the user process and the device driver.

Block Device Drivers

Block device drivers require more involvement from the operating system to perform their tasks. Part of this reason for this additional involvement is because block devices transfer data in fixed-size blocks, and are usually capable of random access. (The device does not need to be capable of random access.)

In a block driver, data transferred between the kernel and the device resides in BSIZE blocks of memory that are managed by buffer headers; together, the blocks of memory form a linked list. Each buffer header is an instance of the *buf* data structure. A field in the buffer header points to the data blocks and to the next and last elements of the linked list. Other fields provide status information. A complete description of the *buf* structure is provided in Chapter 3, "Block Device Drivers."

The buffer mechanism simplifies the overhead on a driver. Because the buffers are provided in the system, specific allocation and memory management routines are not necessary. In addition, the system transfers the data to and from the device.

Special kernel routines are provided for block drivers that handle each function described in the previous list.

Frequently, block devices support character interfaces for performing *raw I/O* in which data is transferred between a user program and the device without relying on a buffering mechanism. Raw I/O is typically used by disk backup programs that copy data sector by sector from a disk.

The two factors that distinguish block I/O from character I/O are:

- The size of data-transfer requests from the kernel to the device is always a multiple of the system-block size (called BSIZE), regardless of the size of the user process' original request. A single user-process request can generate many system requests to the driver. BSIZE is defined in *sys/fs/s5param.h* and varies by file system size. The device's physical block size may be smaller than BSIZE, in which case the device driver initiates multiple physical transfers to transfer a single logical block.

- Transfers are never done directly into a user process' memory area. They are always staged through a pool of BSIZE buffers, commonly known as the buffer cache. Program I/O requests are satisfied directly from the buffers. System V instructs the device driver to read and write from the buffers as necessary. The kernel manages these buffers to perform services such as blocking and unblocking of data and disk cache access.

Differences Between a Driver and an Application Program

Writing a device driver is quite different from writing an application program. When you write a driver, you are confronted with different methods for writing the code, the routines are different, and the method for making the driver "executable" is different. About the only similarity is that both a driver and an application are written in the C programming language. Writing a driver is a complex task. Not only do the differences make the task difficult, but the tools for creating a driver demand a great deal of innovation and persistence.

A device driver varies from an applications program in these ways:

- *Output file* — An application is compiled into an executable *a.out* file that has boundaries that are observed by the kernel when the program executes. Similarly, the whole kernel is an *a.out* file and the driver is only a part of this larger file. When a driver is compiled, it is only taken as far as an object file. When the kernel is linked, all the drivers and the other parts of the kernel such as the scheduler, the swapper, and so on, are collected together to form the new kernel.

- *Format* — An application is a series of routines subordinate to a **main** routine. A driver, conversely, does not have a **main** routine. The kernel acts as the **main** routine and a driver acts as a subroutine of the larger kernel "program."

- *Execution* — An application is executed sequentially by a single process. Because a driver is a part of the kernel and is always available, it is executed by many processes. Driver code must be written to be able to respond to many requests. While applications create an image of the executable image for each caller, kernel requests work with one image of the kernel that is not replicated.

- *Routine execution* — Subroutines that comprise an application program are executed from user space sequentially by instructions in the **main** routine and routines subordinate to it. Most of the routines in a driver are executed on demand when system calls are executed from user programs. After a driver is installed and a device file created to address the driver, a user program opens the device file. The kernel takes the open request and passes it to the driver. The driver contains a routine for opening the device. The driver's open routine contains the code you define for opening the device and for initializing access to the device. Similarly, a driver contains routines for reading, writing, I/O control commands, and for closing access to the device being addressed.

- *Timing* — An application program executes its system calls and routines without having to be concerned with how long an instruction takes to execute or having to be concerned with the interaction of interrupts. In a driver, each kernel routine can affect the timing of a driver, especially kernel routines that write to the console like **cmn_err**(K) and **printf**(K). In addition, the driver timing can even change when routines such as these are removed.

- *Interrupts* — In a driver, if the device for which the driver is written has interrupts, you should provide two mechanisms for handling interrupts. The first is an interrupt handling routine that you write. Secondly, you should protect code that must not be interrupted by surrounding the *critical code section* with the necessary **spl**(K) kernel routines. These kernel routines let you selectively mask out interrupts from other devices.

- *Re-entrant processing* — On a multitasking system like System V, the kernel is capable of tracking many processes at the same time. Each individual process has its own local variables; hence, device driver code should always be reentrant. This means that the driver must be capable of being invoked again before the previous request has been satisfied. For instances when kernel execution must be limited to a single process, see the discussion of interrupt support routines in this chapter.

Special Device Files

To a System V user, a device can be treated like a *file*. A file consists of an ordered sequence of bytes. Files that contain data are called *regular files*, and files that represent devices are called *special device files*. Each file has at least one name; the names of special device files are, by convention, placed in the directory named */dev*.

Each special device file has a *device number* that uniquely identifies the device. The device number consists of two parts, the *major number* and the *minor number*. The major number tells the kernel which device driver will handle requests for this special file. The minor number can be used by the driver to provide more information about a particular unit of the devices that it controls (such as the unit number). For example, all the ports on an eight-port serial card have the same major device number, but they would have eight separate minor device numbers. The minor device number part of the device number usually encodes the unit number. However, a device driver can dedicate some of the bits in the minor number to indicate special options, such as to use double density in the case of a floppy disk.

Examining the Device Number

Before the user process can request input or output, the process must first have opened a special device file. A special device file looks like an ordinary disk file, except that it was created by the utility program, **mknod**(C), described in the *User's Reference* manual. The file appears in a directory and has owner and permission fields, as does any regular file, but it contains no file size data. Instead, it has the *major* and *minor* numbers associated with it. The **ls -l** command displays numbers like these:

```
brw-------  2 sysinfo   sysinfo  1, 15 Aug 21 05:34 /dev/hd01
crw-rw-rw-  1 bin       bin      5,  0 Aug  7 18:20 /dev/tty1a
crw-rw-rw-  1 bin       bin      5,  1 Aug 10 17:33 /dev/tty1b
```

Here the */dev/tty1a* file has a major device number of 5 and a minor device number of 0. */dev/tty1b* has a major device number of 5 and a minor device number of 1. The */dev/hd01* file has a major device number of 1 and a minor device number of 15.

Accessing a Device

When a user process opens the special device file, the kernel recognizes that it is a special device file and uses the major number to index a table of entry points. If the special device file designates a character device, it uses the *cdevsw* table; if it designates a block device, it uses the *bdevsw* table. These two tables are defined in the *sys/conf.h* file.

When a user process uses the **open**(S) or **fopen**(S) system service on a file, the kernel calls the device driver's open entry-point through the *cdevsw* or *bdevsw* table, supplying the device number (both the major and minor numbers) as an argument. Upon entry to the **xxopen** driver routine, the driver uses the **minor**(K) macro to extract the minor number from the device number.

These special device files should have meaningful names and should reside in the */dev* directory.

Kernel Environment

This section briefly discusses a few functional aspects of the operating system: modes of operation, context switching, system-mode stack use, task-time processing, and interrupt-time processing. It also describes the services provided to device drivers by the kernel, and the rules that device drivers are required to obey.

What is an Interrupt?

An interrupt is a signal from a device that tells the kernel that an action has been completed or that the sending process or device requires immediate attention. The System V kernel depends on interrupts to schedule processing efficiently. Interrupts are processed by a programmable interrupt controller (PIC) on the mother board. When a device is ready to perform an action, a signal is sent to the CPU.

Modes of Operation

When a process is executing instructions in the user program, it is said to be in *user mode*; when executing kernel instructions, it is said to be in *system mode* or *kernel mode*. An interrupt causes the kernel to switch to system mode if it was in user mode and passes control to the interrupt routine of the appropriate device driver. After processing the interrupt, control of the process returns to the kernel picking up processing where it was at the time the interrupt occurred. The code that was executed to handle the interrupt is called *interrupt-time processing*. All other processing, execution in user programs, and execution in the kernel resulting from system calls, is called *task-time* processing.

Although all processes originate as user programs, a given process may run in either user or system mode. In system mode, a process executes kernel code and has privileged access to I/O devices and other services. In user mode, a process executes users' program code, and has no special privileges. In fact, System V provides a high level of protection for processes in user mode to prevent a program from inadvertently damaging the system or other programs. A process voluntarily enters system mode when it makes a system call. If an interrupt or trap is received while a process is executing in user mode, the process will switch into system mode to handle the interrupt.

Upon return from an interrupt to user mode, the process may lose control of the CPU, and the kernel may decide to switch control, or *context* (described in the following section), to a different process.

Context Switching

Context switching occurs when the kernel transfers control of the CPU from the currently executing process to a different process.

The kernel makes a context switch whenever:

- The process' time slice expires (only in user mode).
- The user process makes a system call that cannot be completed immediately, as in the case of a read from a slow input device, such as a disk or a tape. When this happens, the device driver may call the kernel routine **sleep**(K).
- An interrupt is received that lets a sleeping process continue. This case will occur when the process has been sleeping at high priority, waiting for the interrupt handler to call **wakeup**(K) to indicate a completed I/O request. If the priority at which the process is sleeping is higher than that of the currently running process, a context switch occurs.

In system mode, a context switch is always voluntary, by way of a call to the **sleep** routine. Interrupts can still arrive while the kernel is in system mode (they can be locked out for short periods of time, if necessary), but when the interrupt-service routine returns, control passes back to the interrupted process.

The System-Mode Stack

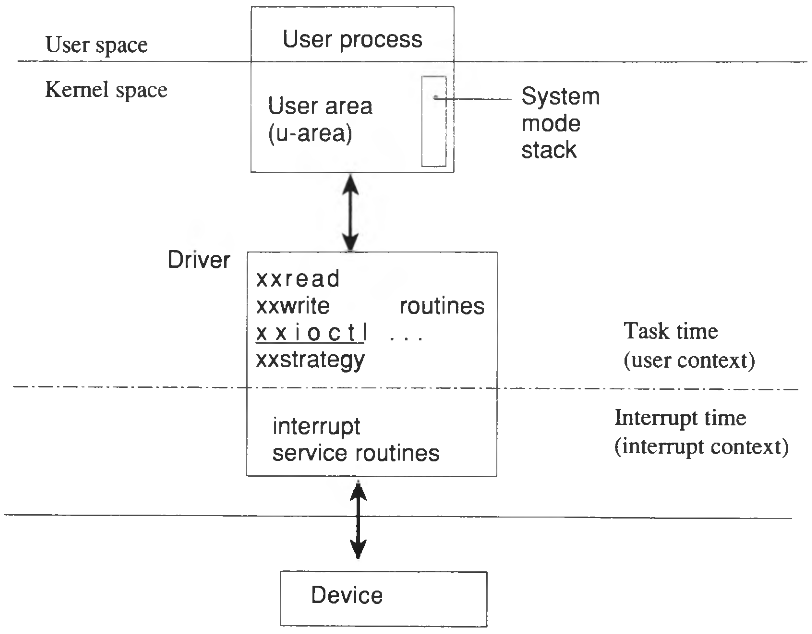
Each process has a special area of memory associated with it, called the *u-area*. The u-area is not directly accessible to a user process (that is, it is not in the process' user address space). It contains the information the kernel needs to manage the process while it is running, and contains space for a system-mode stack. The u-area is an instance of the *user* structure defined in *sys/user.h*.

When any process makes a system call, its registers are preserved in its u-area, and the stack pointer is moved to the beginning of its system-mode stack area. When the system call has completed, the registers are restored from the u-area, the stack pointer is restored to the process' stack, and control is returned to the process. Since each process in the system has its own u-area, a system running *n* processes has *n* user stacks and *n* system stacks.

The System V operating system (and therefore the task-time portions of the device drivers) uses a fixed-size, kernel stack in the u-area. In System V, the size of this per-process stack is 4096 bytes.

Interrupt service routines (**xxintr** and subordinate routines) make use of whatever system stack was set up at the time of the interrupt. If the interrupt occurs while the currently running process is in user mode, the interrupt-service routine will have the entire kernel stack area for its use. However, if the interrupt takes place while the process is in system mode, the interrupt-service routine will be sharing the kernel stack area. For this reason, interrupt-service routines must minimize their frame-variable declarations, keeping their frame requirements to as few bytes as possible. In addition, all interrupt routines should be kept as small as possible. This is especially true when an interrupt routine is servicing many interrupts. The longer the interrupt routine, the greater the chance that an interrupt will be missed.

The following diagram depicts the relationship of the stack, task-time and interrupt-time processing:



Task-Time Processing

The operating system manages a number of processes, each corresponding to a user program. Any particular process may be running in system mode or user mode at any given time. When a process makes a system call to request a kernel service, the process switches to system mode and starts running kernel code. When the kernel is executing code at the request of a user program, it is doing *task-time processing*.

Each time a driver is invoked, it services only the specific system call that the user process requested. The active process' u-area is mapped into the kernel's address space during task-time, so when kernel code is executing it has information about the request and process that it is serving.

Often the kernel cannot service a request immediately. The request may require I/O, or the request itself could be an instruction to wait a while. When a process in system mode sleeps (blocks), awaiting some event, the system scheduler schedules some other process, which may be in either user or system mode. The system continues operations but switches from the execution of a sleeping process to an active one.

I/O requests from user processes are passed by means of system calls to the device driver. Some parameters of the request, such as byte count and transfer address, are kept in the u-area. The task-time portions of the driver can reference and perhaps modify the u-area, since the currently running process' u-area is mapped into the kernel's address space at that time.

Interrupt-Time Processing

When a device interrupt is received, the tasks performed as a result of the interrupt are referred to as *interrupt-time processing*. When an interrupt arrives, any of the active processes on the system may be executing. That is, the system may be running in the context of any current active process. This process may or may not be the process that is expecting the interrupt. In fact, it is highly unlikely that the currently running process will be the process expecting the interrupt.

Even if the incoming interrupt signals the completion of a user process' request, the interrupt-service routine can take no direct action. Typically, a process will be asleep, waiting for I/O, and the interrupt from the device indicates that the I/O request is complete or that data is ready to be transferred. The interrupt routine needs to transfer the data to kernel buffers and wakeup the user process. Then, at task time, the data can be transferred to the user process. Any data or status that the interrupt-service routine wants to return to the task-time portion of the driver (and hence to the requesting user program) must also be passed by means of static variables.

The task-time portion of the device driver keeps the automatic variables in its system-mode stack, which is in the u-area. This u-area is not mapped into the kernel's address space at interrupt time. In this case, the u-area there belongs to another process. The correct u-area might even be out on the swap disk. Thus, the interrupt-service routine must never attempt to store data in the u-area or in user memory, and the I/O device itself must not transfer directly into the user's memory area. An interrupt routine can make no assumptions about the u-area.

Usually, this is not a problem. Character devices typically make use of small, system-supplied buffers called *character lists* (*clists*). Block devices use BSIZE buffers in the system-buffer pool. The task-time portion of the driver transfers the data from the buffers into user memory as follows:

- Typically, the task-time portion of the device driver issues a **sleep**(K) call after it makes the initial I/O request.
- The interrupt-service routine (**xxintr**) must decide if an interrupt is valid and any action to be taken as a result of the interrupt. The interrupt routine must be able to decide if it needs to notify the task-time portion of the driver as opposed to issuing another I/O command.
- If the task time portion of the driver should be notified, the interrupt routine puts any status information into static data and issues a **wakeup**(K) call to the task-time portion.
- The interrupt-service routine then returns to the operating system, which in turn returns control to the interrupted context.
- The system scheduler eventually reschedules the running process so that the newly awakened process is executed.
- The task-time portion of the device driver finds that it has returned from the **sleep** call and that there are status and data bytes waiting in static variables.

Access to static variables that can be modified at interrupt time is interlocked with the **spl**(K) system priority level routines. These routines raise the interrupt priority of the CPU so that interrupts that might cause data or a data structure to change are locked out until the **splx**(K) routine is called. This period must be kept as short as possible.

Device drivers that use the standard interfaces to the kernel have a method for passing information between the interrupt-time portion of a driver and the task-time portion. Standard I/O device drivers for block devices note the outcome of the data transfer in the buffer headers associated with the transfer. The header for the list of transfers that the driver is working on is defined in *sys/iobuf.h*. The header for the buffer associated with the current transfer is defined in *sys/buf.h*. Standard character I/O device drivers use the per-device tty structure (defined in *sys/tty.h*) to pass information about the I/O request. The *tty* structure is described in Chapter 8, "Line Disciplines."

Interrupt Routine Guidelines

An interrupt service routine operates in a more restricted environment than a task-time routine, since it cannot make any assumptions about the state of the system or about the presence of particular user processes or user data in system memory.

The key things to remember are that the user process is mapped into memory, and its u-area is mapped into the kernel's address space, only at task time. Task-time processing occurs whenever the user-program code is executing (*user mode*) or the operating system is executing and performing services for the program (*system mode*).

Do not assume that the u-area is mapped into memory during the execution of an interrupt routine. No interrupt routine, nor any routine that may be called at interrupt time, may make any reference to user memory, the u-area, or nonstatic memory locations. This means that the task-time portion of the driver must not try to pass addresses of its stack (automatic) variables and buffers to devices and interrupt-service routines. Those locations are valid only when that individual user process is executing.

Memory Management

This section describes the many memory management techniques available in System V. Much of the information is taken from the section (K) manual pages and is grouped together by purpose. Topics described in this section are:

- Allocating Physical Memory at Initialization Time
- Allocating Dynamic Memory
- Designating an Address for Memory-Mapped I/O
- Memory Shared With User Processes

Physical Memory Allocation

Drivers that require physically contiguous memory must allocate it when the system is brought up to ensure adequate supply. The **memget**(K) routine is provided for this purpose.

The memget(K) Routine

The **memget** routine is used to obtain permanent, contiguous memory for a driver at initialization time. It is intended for memory that the driver will always have and use. Its argument is the size of memory in pages. Use the macro **btoc**(K) to calculate the number of pages from the number of bytes required. **memget**'s return value is also in pages, so the **ctob**(K) macro must be used to translate the return value of **memget** into a kernel virtual address. Both **ctob** and **btoc** are defined in the file *sys/sysmacros.h*.

NOTE: This routine is intended for use in a driver's initialization routine (**xxinit**) for use before any user processes have been run. Calling **memget** in other routines can result in a caller sleeping forever. If physically contiguous memory is not immediately available, **memget** goes to sleep with periodic checks, but never rearranges pages to obtain the memory. Thus if the memory is not available during a check, **memget** returns to sleep, and may never find available memory.

The return value is a page frame number of the first frame of memory allocated.

The memget(K) Routine Syntax

```
int
memget(pages)
int pages;
```

pages is the number of pages to allocate.

To obtain a permanent 4K buffer for a driver, use the following code statement:

```
char *always;

always = (char *) ctob( memget( btoc( 0x1000 ) ) );
```

Allocating Dynamic Memory

In the course of a driver's activity it must allocate and release memory for use as buffers and local storage. The **sptalloc**(K) and **sptfree**(K) routines are provided for allocating and releasing dynamic memory.

The sptalloc(K) Routine

The **sptalloc** routine is used to obtain temporary memory for use by device drivers, or to map a device into memory for memory mapped I/O. Memory is obtained from the system's virtual memory pool. When the driver is through with the memory, the memory should be released via **sptfree**(K). This routine returns a virtual address usable by any kernel or driver routine.

Memory allocated by **sptalloc** is not physically contiguous. **sptalloc** handles all links between segments. The memory allocated is never swapped out, and it only belongs to the driver that allocated it until the memory is freed with **sptfree**(K).

The usual way to call **sptalloc** is as follows:

```
vaddr = sptalloc(pages, PG_P, 0, 1);
```

Where **vaddr** is the returned virtual address, *pages* are the number of requested pages, PG_P indicates "page present," **0** (zero) indicates that requested memory is taken from the kernel memory pool, and **1** indicates to return immediately if memory is not available.

Because **sptalloc** may sleep, it should not be used at interrupt time (**xxintr** routine).

This routine returns the kernel virtual address of the memory allocated. NULL is returned if map space is not available. The size of the map is determined by the constant **sptmap** which is configurable using the Link Kit.

Although **vasbind**(K) provides a more generalized method of sharing memory between the kernel and a user process, **sptalloc** with *mode* set to PG_P | PG_RW | PG_US may be used instead of **vasbind**, but the results are different.

A mapping performed with **vasbind** creates a region of memory shared only between the kernel and a specific user process. **sptalloc** creates a mapping accessible by the kernel and *all* processes. However, only those processes that have been told the virtual address returned from **sptalloc**, will know the address at which to access the memory.

The sptalloc(K) Routine Syntax

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

caddr_t
sptalloc(pages, mode, base, flag)
int pages, mode, base, flag;
```

The parameters are:

pages the number of requested pages

mode page descriptor table entry field mask. Possible values are defined in *sys/immu.h* and are:

- PG_P — page-present bit. This flag must be present for driver use. PG_P causes the *present* bit to be set in the page table entry. The CPU uses the present bit to differentiate between pages that have to be faulted in and pages that are already there.
- PG_RW — make segment usable for either reading or writing. If this flag is not ORed into *mode*, than the segment is read-only. This flag only has meaning when used with PG_US to indicate if a user can access the segment for both reading and writing. (Kernel processes can read or write any present page whether write access is "permitted' or not.)
- PG_US — identify owner of memory. If ORed in, memory is allocated for a user process, if omitted, memory is for a kernel process. If selected, any user process can access the page. Use with PG_RW if write permission is required; without PG_RW, the page is read-only. To use this capability, a driver must pass the return value from **sptalloc** back to the user process for it to "know" where the memory is, but this doesn't limit its use to that process.

base set to 0 (zero) to allocate from previously allocated kernel memory, or set to an physical address pointing to previously allocated memory elsewhere.

flag	Set to 1 to return immediately if memory is not available. Set to 0 (zero) to sleep until memory is available. If only one page is being requested, and memory is not available, sleep occurs however *flag* is set. When sleeping is requested, **sptalloc** sleeps with a priority of 0 (zero) and is not affected by signals.

The sptfree(K) Routine

The **sptfree** routine frees memory obtained from **sptalloc**(K). The arguments are the pointer returned by **sptalloc**, the size of the memory (same as passed to **sptalloc**) and a flag which denotes whether you want this freed memory to go back into the free page list. For drivers which use this to free memory obtained from **sptalloc**, the flag must always be 1.

For example, to release the memory obtained by the **sptalloc** above, and free it completely, use the following statement:

```
sptfree(va, npages, 1);
```

sptfree(K) Syntax

```
void
sptfree(va, npages, freeflg)
char *va;
int npages;
int freeflg;
```

The argument *va* is the virtual address returned from a previous call to **sptalloc**.

The value of *npages* is the number of pages to free. This should be the same number of pages allocated by a previous call to **sptalloc**.

The argument *freeflg* indicates whether to actually free the memory pages or not. If *freeflg* is not set, the memory pages are not freed. This is used, for example, when freeing memory-mapped I/O space.

Memory-Mapped I/O Allocation

The **sptalloc**(K) is used to allocate an address for memory-mapped I/O. Refer to the description in the last section for more information about **sptalloc** and its companion routine **sptfree**.

To use **sptalloc** for allocating memory-mapped I/O, use **sptalloc** with the following format:

```
vaddr = sptalloc(1, PG_P, physical-address, 1);
```

This requests one page that is present in the kernel at *physical-address* (the *base* argument), and that **sptalloc** return immediately if memory is not available.

To release a memory-mapped I/O space, use **sptfree**(K) with the following format:

```
sptfree(virtual-address, 1, 0);
```

This releases one page, but does not actually free the memory page from kernel memory.

Allocating Memory to Share With User Processes

The **vas** routines are provided for allocating memory to be shared with user processes. This type of memory allocation is frequently used for video adapter drivers so that user processes can access memory for creating screen displays.

The vas(K) Routines

These routines allow a driver to map physical memory so that it can be read from or written to by both a driver and a calling user process. These routines are generally used to allow user processes to directly access video adapter memory. Memory that has been mapped using these routines is visible to the kernel and to a calling process. However, the mapping is not globally visible to all processes.

vasmalloc allocates virtual memory. Use this routine to obtain virtual address space that is not currently in use. **vasmalloc** can only allocate four megabytes of virtual address space on each call. Requests less than this amount are rounded up to four megabytes; requests larger than this amount cause an error to occur. **vasmalloc** returns an address to virtual user memory; no actual physical memory is allocated by this routine. The *nbytes* argument can be specified as 1 to allocate four megabytes, but 0 (zero) or not specifying this argument is not permissible.

vasbind binds a specified virtual address to a physical address. This routine ensures that a problem will not occur with the bound memory being swapped out and causing a page fault and panic in the kernel. Before using **vasbind**, call **vasmapped** to determine if memory has already been mapped for the calling process.

The physical address supplied to **vasbind** may be the address of an I/O address space, for example, a memory-mapped I/O address. Or specify -1 to request that the memory be allocated from the kernel free memory pool.

When **vasbind** completes, the driver must pass the virtual address back to the user process using **copyout**(K) or another similar routine. Calls to **vasbind** must not specify an address in the text, data, or shared data segments of a user process.

The upper limit for user virtual memory is set at KVBASE; above this is the kernel virtual address space. The virtual address supplied to **vasbind** must be in user virtual memory (below KVBASE), and must not be in use by the current process.

vasmapped determines if a mapping is already in place.

vasunbind undoes a mapping.

These routines cannot be called from a driver's interrupt routine (**xxintr**).

vasbind returns -1 if an error occurs or if an error is found in *u.u_error*. **vasmalloc** returns a virtual address. **vasunbind** returns -1 if an error is found in *u.u_error*, or if the virtual address could not be found. **vasmapped** returns the virtual address at which the supplied physical address is bound, or 0 (zero) if the physical address is not bound.

The vas(K) Routine Syntax

```
int
vasbind(paddr, vaddr, nbytes)
paddr_t paddr;
caddr_t vaddr;
unsigned int nbytes;

caddr_t
vasmalloc(paddr, nbytes)
paddr_t paddr;
unsigned int nbytes;
```

```
caddr_t
vasmapped(paddr, nbytes)
paddr_t paddr;
unsigned int nbytes;

int
vasunbind(vaddr, nbytes)
caddr_t vaddr;
unsigned int nbytes;
```

Parameters:

nbytes	number of bytes of memory to allocate, bind, or unbind. For **vasmalloc**, *nbytes* can be specified as 1 to allocate four megabytes, but 0 (zero) or not specifying this argument is not permissible.
paddr	Physical address at which the specified virtual address is to be bound. When calling **vasbind**, *paddr* can be set to -1 to indicate that the requested user virtual memory is to be allocated.
vaddr	Virtual address to bind or unbind to or from physical memory

Direct Memory Access (DMA)

This section describes how to effectively use DMA routines for transferring data with your system's Direct Memory Access (DMA) controller. After the discussion on using the DMA routines, a section is provided describing the syntax and purpose of each DMA routine.

DMA provides a means of offloading disk I/O from the CPU to free it for other tasks. DMA consists of a controller chip set that conveys data between memory and a disk controller. The DMA controller contains separate channels that are allocated, used for I/O transfer, and then released. Because a finite number of channels exist, their use must be managed effectively. Channels are identified by the number of bits in the data path, either 8 or 16 bits. System V provides four 8-bit channels and three 16-bit channels. The channel selected for your device depends on the capabilities of the device itself. A driver selects a channel by hardware dependencies and by avoiding using a channel already reserved for use by another device.

Refer to *sys/dma.h* for channel and structure definitions used when performing DMA.

Allocating Memory for DMA Physical I/O

When performing DMA during physical I/O, contiguous physical memory must be allocated for storing the data being transferred to the device and also for data that is being transferred from the device. The kernel provides the **db_alloc**(K) routine for allocating contiguous physical memory. The **db_free**(K) routine is used to release physical memory. **db_alloc** cannot be used in either an initialization routine, or an interrupt routine. Use **memget**(K) to allocate contiguous physical memory in an initialization routine. Refer to either the "Memory Management" section in this chapter, or the section (K) manual pages for more information on these routines.

The db_alloc(K) and db_free(K) Routines

The **db_alloc** routine allocates one block of physically contiguous memory. Contiguous memory is necessary for performing DMA transfers. Memory for all other uses should be allocated using standard memory allocation routines for your machine.

db_free releases the previously allocated memory.

The db_alloc(K) and db_free(K) Routine Syntax

```
int
db_alloc(dv)
struct devbuf *dv;

int
db_free(dv)
struct devbuf *dv;
```

dv points to an instance of the *devbuf* structure.

The *devbuf* structure is:

Type	Field	Description	
paddr_t	bufptr;	/* pointer to start of buffer	*/
paddr_t	bufend;	/* pointer to end of buffer	*/
long	size;	/* size of buffer	*/
paddr_t	head;	/* put buffer data here	*/
paddr_t	tail;	/* get buffer data here	*/

Set *size* to the block size before calling **db_alloc**. All other fields in the structure are read only.

db_alloc must not be used during the driver's initialization function. The **memget**(K) function can be called to obtain contiguous memory during driver initialization. Reading from and writing to memory areas allocated using **db_alloc**(K) must be performed using the **db_read**(K) and **db_write**(K) routines only. **db_read** conveys data from the allocated memory to a user process, and **db_write** conveys data from a user process to the allocated memory.

The **db_alloc** routine returns zero (0) if no memory is available; otherwise, 1 is returned. **db_free** always returns zero (0) for normal completion.

The following example allocates a single 120K buffer:

```
struct devbuf dv;
dv.size = (long) (120 * 1024)  /* 120 times 1K */
if (db_alloc(&dv) == 0) {
        cmn_err(CE_NOTE, "db_alloc failed");
        return(-1);
}
```

The following example releases previously allocated memory:

```
struct devbuf dv;
db_free(&dv);
```

When transferring data from the allocated memory, use **db_read**(K) to transfer the data to user space, and **db_write**(K) to transfer data from user space. Refer to section (K) for more information on these routines. The **db_read** and **db_write** routines can only be used for transferring data during physical I/O. Use at any other time causes a panic.

DMA Routines Provided In the Kernel

The kernel provides the following routines for DMA:

- **db_alloc**(K) — Allocate contiguous physical memory for a DMA transfer
- **db_free**(K) — Release allocated memory
- **db_read**(K) — Transfer data from allocated memory to a user process
- **db_write**(K) — Transfer data from a user process to allocated memory
- **dma_alloc**(K) — Allocate a channel for non-queued DMA
- **dma_breakup**(K) — Break up a physical I/O request into block-size units
- **dma_enable**(K) — Perform a DMA transfer
- **dma_param**(K) — Enable a DMA channel for access
- **dma_relse**(K) — Release an allocated DMA channel
- **dma_resid**(K) — Return the number of bytes not transferred
- **dma_start**(K) — Allocate a channel for queued DMA

These routines are described in the section (K) of manual pages in this manual and their use is summarized in the sections that follow.

DMA Transfer

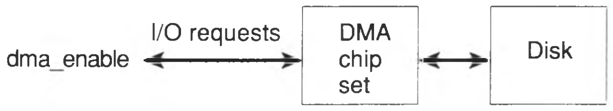
Two methods of DMA transfer are provided in the kernel:

- Managing DMA requests in a driver. This method requires that you allocate a channel with **dma_alloc**(K). You have the choice of using **dma_alloc** either at task time in which processes unable to get a channel sleep until a channel is available, or at interrupt time so that an immediate return occurs if a channel is unavailable.

- Queueing DMA requests so that they are executed whenever the requested channel is free. This method is performed with the **dma_start**(K) routine. **dma_start** requests that a channel be allocated, and when available, calls the driver's **xxd_proc** routine that contains other DMA kernel routines for enabling access to the channel and for transferring data between memory and the DMA controller. This method requires that the driver provide an instance of the *dmareq* structure in which to set parameters to describe the transfer. When the transfer is complete, the driver then releases the channel. The queueing mechanism can be used at task time or at interrupt time.

Managed DMA (Non-Queued)

This form of DMA transfer is summarized in the following diagram:



After a channel is allocated using **dma_alloc**, the **dma_param** routine is used to enable access to the channel. Then the actual DMA transfer is performed by the **dma_enable** routine with the DMA chip set. When done, the **dma_relse** routine is called to release the channel. The DMA chip set interacts with the disk to move data to and from it directly.

The steps required to perform this type of DMA transfer are as follows:

- **dma_alloc**(K) is called to allocate a DMA channel to the driver.

- Once the channel has been allocated, the driver must set up the parameters of the transfer by calling **dma_param**(K) with the appropriate parameters.

- After the parameters have been set, the driver begins the actual transfer by calling **dma_enable**(K).

- After the transfer is complete, the routine **dma_resid**(K) may be called to find out the amount of data that was not transferred.

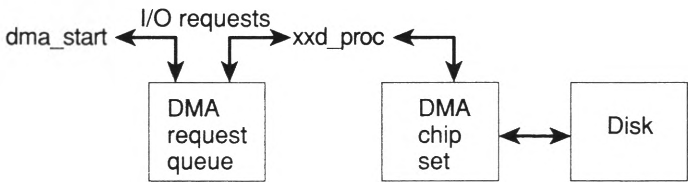
- Another DMA transfer must be initiated to complete the request. Once the transfer is completed, the driver must call **dma_relse**(K) to release the channel for use by other drivers.

The following routine demonstrates the use of the DMA routines. It accepts a buffer and a byte count, and writes the data in the buffer to DMA channel 1 (DMA_CH1):

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

alicia_dma( buf, count )
paddr_t buf;
long count;
{
        long leftover;

        if ( dma_alloc( DMA_CH1, DMA_BLOCK ) == 0 ) {
                printf( "Error: couldn't allocate DMA channel" );
        } else {
                dma_param(DMA_CH1, DMA_Wrmode, buf, count);
                dma_enable();

                /* driver must now wait for the device to signal  */
                /* that the DMA request is complete.  This can be */
                /* done via an interrupt, or status register      */

                /* driver routine that waits for signal from device */
                wait_DMA();

                leftover = dma_resid();
                if ( leftover > 0L ) {
                        printf("Error: DMA request not completed, ");
                       printf("%ld bytes not transferred\n", leftover);
                }
                dma_relse( DMA_CH1 );
        }
}
```

Queued DMA Transfer

Using the kernel's DMA request queue requires a slightly different procedure than given in the last transfer discussion. The following diagram depicts queued DMA:



The procedure is as follows:

- To submit a request, the driver routine calls **dma_start**(K), passing to it a *dmareq* structure defining the DMA request. The driver's routine must have initialized the structure with the following information:

  d_chan: channel to perform the request
  d_mode: direction of the transfer (read or write)
  d_addr: physical address from which or to which to transfer
  d_cnt: number of bytes to transfer
  d_proc: address of the routine to do the transfer
  d_params: parameter to the routine pointed to by d_proc

- The *d_proc* element should point to the routine to be called by the kernel when it is time to service this particular request. This routine will be called with the DMA channel already allocated, so it should call **dma_param**, **dma_enable**, **dma_resid** if desired, and **dma_relse**. The routine should be as short as possible, since it may be called during another driver's interrupt routine. When this service routine is called, the kernel also passes to it a single argument, a pointer to the *dmareq* structure given by the call to **dma_start**. This pointer is then used to get the particulars of the DMA request.

- Note that the service routine must call **dma_relse** to release the DMA channel.

The following two routines demonstrate how to queue and service a DMA request using the kernel's DMA queue:

```
#include "sys/cmn_err.h"
#include "sys/dma.h"

static long leftover;     /* number of bytes not     */
                          /* transferred in request */

queue_dma( buf, count )
paddr_t buf;
long count;
{
        struct dmareq dp;
        int service_dma();

        /* set up DMA request structure */
        dp.d_chan = DMA_CH2;
        dp.d_mode = DMA_Wrmode;
        dp.d_addr = buf;
        dp.d_cnt = count;
        dp.d_proc = service_dma;
        dp.d_params = "DMA request from queue_dma";

        /* Queue DMA request.  If the request is completed */
        /* immediately, then return, otherwise, sleep      */
        /* until service_DMA says transfer is complete.    */

        if ( dma_start( &dp ) )
                return(0);
        else {
                /* go to sleep until transfer is completed */
                sleep( &leftover, PZERO+1 );
        }
        if ( leftover > 0L ) {
                cmn_err(CE_CONT,"Error: DMA not completed,");
                cmn_err(CE_CONT,"%ld bytes not transferred\n", leftover);
        }
        return(0);
}
```

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

service_dma( dp )
struct dmareq *dp;
{
  cmn_err(CE_CONT, "Now servicing %s\n", dp->d_params );

dma_param( dp->d_chan, dp->d_mode, dp->d_addr, dp->d_cnt );
dma_enable( dp->d_chan );

/* driver must now wait for the device to signal that */
/* the DMA request is complete.  This can be done via */
/* an interrupt, or status register */

  wait_DMA();
/* driver routine that waits for signal from device */


  leftover = dma_resid();
  if ( leftover > 0L )
    printf( "Error: %s not completed,
             %ld bytes not transferred\n",
              dp->d_params, leftover );
  dma_relse( dp->d_chan );
  /* wake up sleeping requester, if any */
  wakeup( &leftover );
  return(0);
}
```

The routines in this section interface with the DMA controller. They are individually discussed in detail in the (K) reference manual section.

Kernel Routine Summary

This section describes the functional categories for the kernel routines that are provided for driver development. The following list describes how the section (K) kernel routines are used in a driver:

- **Address conversion** — *btoc, btoms, ctob, ktop, ptok, vtop*

- **Block driver routines (buffer management)** — *brelse, clrbuf, disksort, getablk, geteblk, iodone, iowait, paddr, physck, physio*

- **Block driver routines not interacting with buffers** — *btoc, btoms, cmn_err, ctob, db_alloc, db_free, db_read, db_write, deverr, dma_alloc, dma_breakup, dma_enable, dma_param, dma_relse, dma_resid, dma_start, printcfg, scsi_get_gen_cmd, scsi_getdev, scsi_mkadr3, scsi_s2tos, scsi_s3tol, scsi_stok, scsi_stol, scsi_swap4, sleep, spl0, spl1, spl2, spl3, spl4, spl5, spl6, spl7, splbuf, splhi, splx, sptalloc, sptfree, suser, timeout, untimeout, vtop, wakeup*

- **Character driver routines** — *bcopy, btoc, btoms, ctob, bzero, canon, cmn_err, copyin, copyio, copyout, cpass, delay, deverr, emdupmap, emunmap, fubyte, fuword, getc, getcb, getcbp, getcf, getchar, inb, ind, inw, ktop, longjmp, major, memget, minor, outb, outd, outw, panic, passc, printf, printcfg, psignal, ptok, putc, putcb, putcf, putchar, repinsb, repinsd, repinsw, repoutsb, repoutsd, repoutsw, selfailure, selsuccess, selwakeup, signal, spl0, spl1, spl2, spl3, spl4, spl5, spl6, spl7, splbuf, splcli, splhi, splni, splpp, spltty, splx, ttclose, ttin, ttinit, ttiocom, ttiwake, ttopen, ttout, ttowake, ttread, ttrdchk, ttrstrt, ttselect, tttimeo, ttwrite, ttxput, ttyflush, ttywait, vasbind, vasmalloc, vasunbind, vtop, wakeup*

- **Character list (clist) management** — *getc, getcb, getcbp, getcf, putc, putcb, putcf*

- **Clear memory** — *bzero, clrbuf*

- **Copy data between kernel addresses** — *bcopy, copyio*
- **Copy kernel data to user space** — *copyio, copyout, passc, subyte, suword, ttread*
- **Copy user data to kernel space** — *copyio, copyin, cpass, fubyte, fuword, ttwrite*
- **Delays** — *delay, timeout, untimeout*
- **Direct Memory Access (DMA)** — *db_alloc, db_free, db_read, db_write, dma_alloc, dma_breakup, dma_enable, dma_param, dma_relse, dma_resid, dma_start*
- **Device number conversion** — *major, minor*
- **End current system call with error** — *longjmp*
- **Flush the translate lookaside buffer (TLB) before reading from or writing to an I/O port** — *flushtlb*
- **Get a character from console** — *getchar*
- **International character set mapping** — *emdupmap, emunmap*
- **I/O port input** — *inb, ind, inw, repinsb, repinsd, repinsw*
- **I/O port output** — *outb, outd, outw, repoutsb, repoutsd, repoutsw*
- **Memory Management** — *db_alloc, db_free, db_read, db_write, memget, sptalloc, sptfree, vasmalloc, vasbind, vasunbind*
- **Message display** — *cmn_err, deverr, printf, printcfg, putchar*

- **Panic the system** — *cmn_err, panic*
- **Permissions (check for super user)** — *suser*
- **Programmed I/O** — *pio_breakup*
- **select(S) access** — *selsuccess, selfailure, selwakeup, ttiocom*
- **Set priority level (spl)** — *spl0, spl1, spl2, spl3, spl4, spl5, spl6, spl7, splbuf, splcli, splhi, splni, splpp, spltty, splx*
- **Sleep and wake up processes** — *sleep, wakeup*
- **Small computer systems interface (SCSI)** — *scsi_get_gen_cmd, scsi_getdev, scsi_mkadr3, scsi_s2tos, scsi_s3tol, scsi_stok, scsi_stol, scsi_swap4*
- **Signal process(es)** — *psignal, signal*
- **Timing** — *delay, timeout, untimeout*
- **TTY Routines** — *canon, ttclose, ttin, ttinit, ttiocom, ttiwake, ttopen, ttout, ttowake, ttread, ttrdchk, ttrstrt, ttselect, tttimeo, ttwrite, ttxput, ttyflush, ttywait*
- **Video adapter driver routines** — *DISPLAYED, viddoio, vidinitscreen, vidmap, vidresscreen, vidsavscreen, vidumapinit, vidunmap*

Driver Development Considerations

This section provides conceptual information used when developing a driver. The topics are:

- *What To Do and What Not To Do* — A description of the pitfalls and limitations of driver development
- *How Data is Passed To a Driver* — A description of how information is passed from a calling user program to a device driver
- *Sharing Interrupt Vectors* — A description of how to share interrupt vectors between devices.

What To Do and What Not To Do

The following guidelines can help you avoid problems when writing a device driver:

- Make interrupt time processing (**xxintr** routine) as short as possible.
- Protect buffer and *clist* processing with the appropriate **spl** calls.
- Avoid *busy waiting* whenever possible.
- Never use floating-point arithmetic operations in device driver code. Floating-point support is not provided in the kernel.
- If any assembly language device driver sets the direction flag (using **std**), it must clear it (using **cld**) before returning.
- Keep the local (stack) data requirements for your driver very small.
- In interrupt routines, ensure that the **spl** calls are never less than the **spl** level for the device. In addition, use of **spl6, spl7, spltty,** and **splhi** should be kept as minimal as possible and the code they surround should also be kept minimal because these functions block clock interrupts. When the clock is blocked for more than a clock tick, the accuracy of the clock is degraded.

- The *user* structure is read-only except for *u.u_error*, *u.u_count*, *u.u_base*, *u.u_offset*, *u.u_ttyp*, and *u.u_segflg*.

- The *proc* structure is completely read-only.

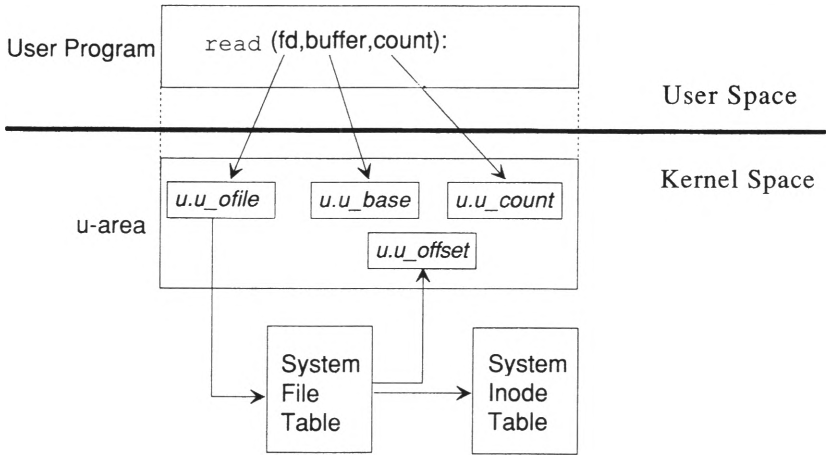
- The *buf* structure is also read-only except for *b_flags*, *b_error*, *b_resid*, and *b_start*. The *b_flags* field must be treated with care in that values can only be ORed in or the same values ANDed out.

- Do not call these section (K) routines in an interrupt routine (**xxintr**): **canon, copyin, copyout, copyio, delay, fubyte, fuword, geteblk, iowait, longjmp, physck, sleep, sptalloc, sptfree, subyte, suser, ttclose, ttiocom, ttioctl, ttopen, ttread, ttwrite, ttywait,** or any of your driver routines that access the *user* structure or call *sleep*.

- When accessing an I/O port with an address above 0x1000, call **flushtlb**(K) to prevent corruption of the I/O address. In addition, protect the **flushtlb** call with **spl7/splx** calls (both **spl** routines are discussed on the **spl**(K) manual page). Refer to the **flushtlb** manual page for an example.

How Data is Passed To a Driver

The task-time portion of the device driver has access to the user's u-area, since this is mapped into the kernel's address space. The kernel routines that process the user process' I/O request place information describing the request into the process' u-area. The following diagram illustrates how u-area parameters are passed to a driver:



The parameters passed in the u-area are:

Parameter	**Contains**
u.u_base	Specifies the address in user data to read/write data for transfer
u.u_count	Specifies the number of bytes to transfer
u.u_offset	Specifies the `start address within the file for transfer
u.u_segflg	Indicates the direction of the transfer. Possible values are 0 to indicate a transfer between the kernel and user data space, 1 to indicate a transfer in kernel space between kernel addresses, and 2 to indicate a transfer between the kernel and user instruction space.

In addition to the parameters passed in the u-area, the kernel I/O routines pass the device number, containing the major and minor device numbers, as a parameter to the driver when it is called. Thus, the driver has all the information it needs to perform the request: the target device, the size of the data transfer, the starting address on the device, and the address in the process' data.

Only device drivers that do not use standard-character and block I/O interfaces in the kernel need to examine the parameters in the u-area. Kernel routines that provide these standard interfaces have converted the values passed in the u-area into values that the driver expects. In the case of the standard block I/O interface, these parameters are set in the buffer header that describes the data transfer.

Device drivers using the standard-character I/O interface use the *clist*-buffering scheme and the routines that manipulate the *clist* to effect the data transfers.

Sharing Interrupt Vectors

I/O devices can only share interrupt vectors if there is a way to poll each device using the shared vector to determine whether that device has posted an interrupt. This feature is hardware dependent and only works when a controller can "float" an interrupt, that is, to keep an interrupt in a neutral state on the bus rather than holding it high or low. Use **configure**(ADM) with the **-T2** or **-T3** options to establish a shared interrupt. See the **configure** manual page for more information.

Sample Drivers

This book is designed to accompany a series of sample working drivers that are provided with your software. The drivers are contained in separate directories located under the */usr/lib/samples/pack.d* directory. The driver directories and files are as follows:

Directory	File	Description
blck	*blck.c*	Floppy disk driver described in Appendix B
	blck.h	Header file
	makefile	Compilation makefile for this driver
exbm	*ev_exbusmouse.c*	Build event structure and queue an event
	event.c	Display events as they enter the queue
	exbmouse.c	Example bus mouse driver
	makefile	Compilation makefile for this driver
exst	*exst.c*	STREAMS driver described in Chapter 9
	makefile	Compilation makefile for this driver
	strtest.c	User program to test *exst.c*
exvd	*exvd.c*	CGA video driver example
	m6845.c	Additional routines for *exvd.c*
	m6845.h	Header file for *m6845.h*
	vidloops.c	Additional routines for *exvd.c*

When examining the video software, refer to the **video**(K) manual page for more information on the video routines. The *exvd.c* driver calls routines that are described on the **video** manual page.

In addition, sample installation scripts are provided in the following directory:

/usr/lib/samples/scripts

Chapter 3

Block Device Drivers

Introduction to Block Devices 3-1

- Character Interface to Block Devices 3-2
- Block Device Driver Routines 3-4
  - Sorting I/O Requests and the xxtab Structure 3-6
- Block Device Driver Routines Descriptions 3-7

Example Block Driver Code 3-14

Introduction to Block Devices

Block devices are those that should be addressed in terms of fixed-size blocks of data, rather than individual bytes. Disks fall into this category, as do some magnetic tape systems. System V file systems always reside on block devices. However, block devices do not have to be used solely in this way.

The kernel maintains a pool of buffers, also called *blocks*, and keeps track of what data is in them, and whether the buffer is dirty (has been modified and therefore needs to be written out to disk). When a user process issues a transfer request to a block device, the kernel buffer routines check the buffer pool to see if the data is already in memory. If it is not, a request is passed to the driver to get the data. The driver only sees fixed-size requests (BSIZE bytes long) coming in from one source, regardless of the size of the process' I/O request. Large requests are broken down into BSIZE blocks and handled individually, since some may be in memory and some may not.

BSIZE is a manifest constant defined in the *sys/fs/s5param.h* file. This constant varies by file system size.

When a process issues a read request, this generally translates into one or more disk blocks. The kernel checks to see which of these is already in memory, and requests that the driver get the remainder. The data from each buffer filled by the driver is copied into the process' memory by the kernel.

In the case of a write request, the kernel copies the data from the user process' memory into the buffer pool. If there are insufficient free buffers, the kernel has the driver write some out to disk using a selection algorithm designed to reduce disk traffic. When all the data is copied out of user space, the kernel can reschedule the process. Note that all the data may not yet be out on a disk; some may be in memory buffers, marked to be written out at a later time.

The steps that occur in block I/O are as follows:

1. The driver gets an empty block. The kernel assigns the address and the block number fields.

2. The **xxstrategy** routine converts the logical block to a physical location on the disk (cylinder, track).

3. Sort the requests for optimal disk performance using **disksort**(K).

4. When **disksort** returns, the **xxstrategy** routine calls **xxstart**. **xxstart** checks to see if the device is busy, if not, takes the next request from the sorted queue, copies the buffer header contents into the controller registers (cylinder, head, start sector, number of sectors, physical address of buffer), and then sends a start-processing command to the device.

5. Wait for an interrupt to signal the completion of the transfer.

6. When the interrupt occurs indicating that the I/O request is completed, signify that I/O is done with **iodone**(K).

7. Release completed buffers by marking them not busy. This is accomplished by ANDing out B_BUSY from *b_flags*. These buffers may be used again by the driver before eventually being returned to the free list. The kernel handles the release and freeing of buffers for a driver.

8. If an error occurred during I/O, send a message to the user process (by setting *u.u_error* which is returned to the user in the *errno* variable) and display a message on the console using **cmn_err**(K).

Character Interface to Block Devices

Sometimes block device drivers provide a character I/O interface as well as one for block I/O. Characters are transferred to and from the device without the use of buffers. The character interface to a driver is called raw I/O or physical I/O because data is read or written without character processing or intermediate buffering using character lists (*clists*) or BSIZE blocks.

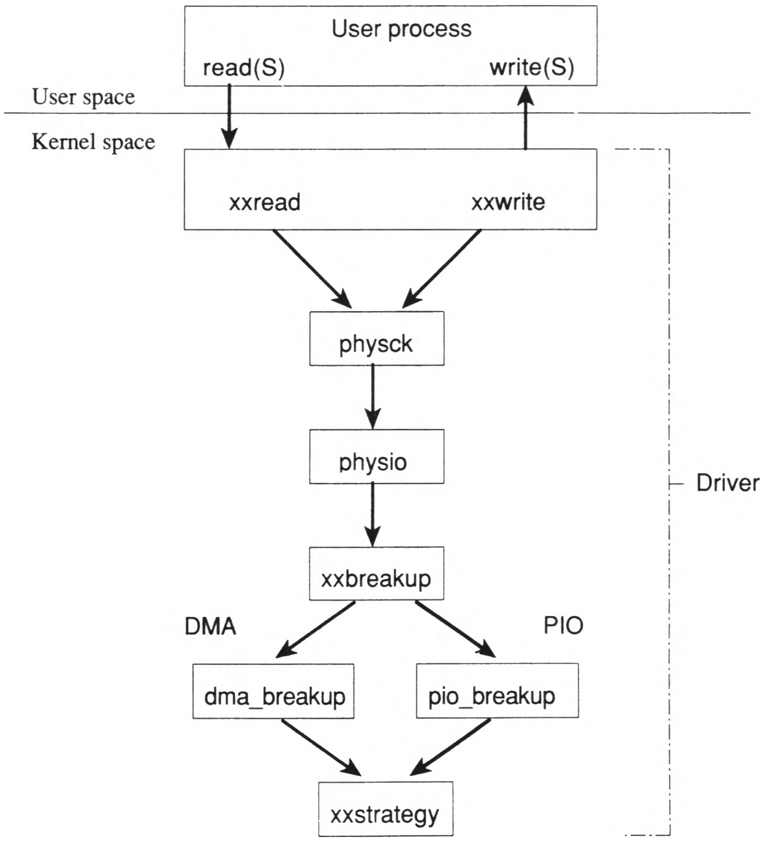
To create a character interface to a block device, a separate special device file is made to access the device through the character device switch table in the kernel. The character device file is created with **mknod**(C) and has the same major and minor number as the block special device file for this device. A block device driver must provide an **xxread** and an **xxwrite** routine to provide entry points to implement character I/O.

When a block device is accessed through a character interface, data transfer takes place directly between the device and the process' memory space. The driver receives the request exactly as the process sent it, for whatever size was specified. There is no kernel support to break the job into BSIZE blocks. Raw I/O has some advantages for certain types of programs.

Programs that need to read or write an entire device can do so more efficiently through the character interface, since the device can be accessed sequentially and large transfers can be used. There is also less copying of data between buffers than is used in the block interface. Thus, disk backup programs, or utilities that copy entire volumes, operate through this interface.

The cost of this extra efficiency is that the process has to be locked in memory during the transfer, since the driver has to know where to buffer the data. **physio**(K), called by the **xxread** and **xxwrite** driver routines, locks the process in memory (core) for the duration of the data transfer.

The following drawing sketches the character interface and how both the **xxread** and **xxwrite** routines use the character interface for accessing **xxstrategy**:



Block Device Driver Routines

This section describes routines that comprise the interface between the kernel and the block device driver. Some of the following routines are supplied by the kernel, and some must be supplied by the driver writer within the device driver.

When writing a block device driver, you should supply the following driver routines:

Routine	*Purpose*
xxopen	Start access to a block device
xxclose	End access to a block device
xxstrategy	Perform buffered I/O with device
xxprint	Display a message on the console
xxread	Perform raw reads on a block device
xxwrite	Perform raw writes to a block device
xxioctl	Perform I/O control commands
xxbreakup	Size I/O request into usable chunks
xxd_proc	Perform queued DMA request
xxinit	Initialize the device when the system boots
xxintr	Handle interrupt from a block device
xxstart	Access device-speci fic I/O ports

The following kernel routines, described in the section (K) manual pages in this guide are used in a block driver:

Routine	*Purpose*
brelse(*bp*)	Release buffer to free list
deverr(*iobuf-ptr,cmd,status,device*)	Display console error message
disksort(*xxtab, bp*)	Add I/O request to queue
dma_breakup(*routine, bp*)	Size DMA data into 512-byte blocks
getablk( )	Get an empty buffer
geteblk( )	Get an empty buffer
iodone(*bp*)	Signal block I/O completion
iowait(*bp*)	Wait for block I/O completion

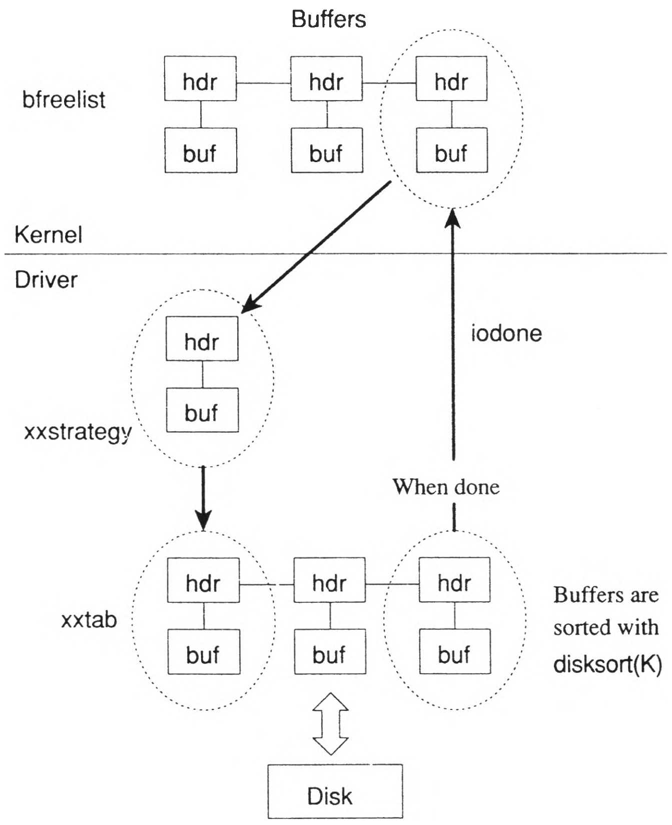
Routine	*Purpose*
paddr(*bp*)	Access buffer data
physck(*nblocks, rwflag*)	See if I/O request is valid
physio(*routine, bp, dev, flg*)	Call *routine* for raw I/O
pio_breakup(*routine, bp, size*)	Size programmed I/O data into specified units

A block device appears to the kernel as a randomly addressable set of records of size BSIZE. The kernel inserts a layer of buffering software between user requests for block devices and the device driver. This buffering improves system performance by acting as a cache, allowing read-ahead and write-behind on block devices.

Each buffer in the cache contains an area for BSIZE bytes of data and has a header associated with it of type *struct buf*, that contains information about the data in the buffer. When an I/O request is passed to the task-time portion of the block device driver, all of the information needed to handle the data transfer request has been stored in the buffer header. This information includes the disk address, and whether a read or a write is to be done. The file *sys/buf.h* describes the fields in the buffer header. The fields most relevant to the device driver are:

Field	**Contains**
b_flags	The buffer status. Always use OR when setting; never clear. Possible values are B_BUSY (buffer is being used), B_DONE (I/O transfer complete), B_ERROR (error occurred during I/O), B_PHYS (buffer header is being used for physical I/O). (Physical I/O is also known as direct or raw I/O.)
b_dev	The major and minor numbers of the device. This field is read-only.
b_bcount	The number of bytes to transfer. This field is read-only.
b_blkno	The block number on the device. This field is read-only.
b_error	The error flag. Set if an error occurred during the transfer.
b_resid	The number of bytes left to be transferred.

The following diagram sketches how an address of a buffer is taken from the free buffer pool (as a result of a call to **geteblk**), used in the **xxstrategy** routine, handed to **disksort**(K) via the *xxtab* structure, and finally released:



The driver validates the transfer parameters in the buffer header, and then queues the buffer on a doubly linked list of pending requests.

Sorting I/O Requests and the xxtab Structure

In each block device driver, a header named *xxtab* (of type *struct iobuf*) points to this chain of requests. The *sys/iobuf.h* file describes the fields in the request queue header. The requests in the list are kept sorted using the **disksort**(K) routine. The device interrupt routine takes its work from this list.

When a transfer request is placed in the list, the process making the request sleeps until the transfer is completed. When the process is awakened, the driver checks the status information from the device interrupt routine. Status is checked by looking at the *b_error* field of the *buf* structure. If the transfer completes successfully, returns a success code to the kernel.

The kernel buffer routines are responsible for correlating the completion of an individual buffer transfer with particular user process requests.

Block Device Driver Routines Descriptions

The interface between the kernel and the block device driver consists of the routines described in the following list:

Syntax: **xxinit( )**

Description: The **xxinit** routine initializes the device when the kernel is first booted. If present, it is called if you place an "i" in the second column table defined in the kernel configuration file */etc/conf/cf.d/mdevice*. The **init** routine should display a message on the console indicating that the associated device has been initialized. (To ensure consistency in **xxinit** routine startup messages, use the **printcfg**(K) kernel routine for displaying the console message.) The **xxinit** routine cannot contain calls to **sleep**(K), or access any *user* structure fields. If **timeout**(K) is called, the actual timing does not commence until the clock is initialized for use. Do not call any of these section (K) routines in an **xxinit** routine: **delay, geteblk, iowait, longjmp, physck, sleep, sptfree**.

Syntax: **xxstart( )**

Description: This routine is used to interact directly with the device's I/O ports using **inb**(K) and **outb**(K). **xxstart** is often called by both task-time and interrupt-time portions of the driver. **xxstart** checks whether the device is ready to accept another transfer request, and when ready, starts the device, usually by sending it a control word.

Syntax:

```
xxopen(dev,flag,id)
dev_t dev;
int flag, id;
```

Description: The **xxopen** routine is called each time the device is opened. This routine initializes the device and performs any error or protection checking.

Parameters: The following values are passed to the driver when the driver routine is called by the kernel:

The value of *dev* specifies the major and minor device numbers.

The values for *flag* may be:

- FAPPEND — open device for appended writes
- FEXCL — open device for exclusive access
- FNDELAY — open device without delay
- FREAD — open device for reading
- FSTOPIO — prevent further I/O (initiated by a security feature)
- FSYNC — open for synchronous writes
- FWRITE — open device for writing

The values for *id* may be:

- OTYP_BLK — open device for block I/O
- OTYP_CHR — open device for raw I/O
- OTYP_MNT — open device for file system mount
- OTYP_SWP — open as swap device

Syntax:	**xxclose(dev, flag)**
	dev_t dev;
	int flag;

Description:	The **xxclose** routine is called on the last close on a device. **xxclose** is responsible for any cleanup that may be required, such as disabling interrupts, clearing device registers, and ejecting media.
Parameters:	The value of *dev* specifies the major and minor device numbers. Refer to the **xxopen** routine for the possible values for *flag*.

Syntax:	**xxstrategy(bp)**
	struct buf *bp;

Description:	The kernel calls the **xxstrategy** routine to queue an I/O request. **xxstrategy** must make sure the request is for a valid block, and then insert the request into the queue. Usually the driver calls **disksort**(K) to insert the request into the queue. The **disksort** routine takes two arguments: a pointer to the head of the queue, and a pointer to the buffer header to be inserted. Before **xxstrategy** is called, the kernel assigns values to these fields of the *buf* structure that are important to a block device driver:

`bp->b_un.b_addr`	(buffer address)
`bp->b_blkno`	(logical block number)

The **xxstrategy** routine converts the logical block to a physical location and then calls **disksort**(K) to sort the requests for optimal disk performance. When **disksort** returns, **xxstrategy** calls **xxstart** to see if the device is busy. If the device is not busy, **xxstart** takes the next request from the sorted queue, copies the buffer header contents into the controller registers (cylinder, head, start sector, number of sectors, physical address of the buffer), then sends a start-processing command to the device.

Parameters: The *bp* argument specifies a pointer to a buffer header.

Syntax:

xxprint(dev, string)
dev_t dev;
char *string;

Description: The kernel calls the **xxprint** routine to display a message on the console when an error occurs on a block device. The routine need only consist of a **cmn_err**(K) call and a mechanism to check for the number of occurrences on a particular device. If a device is causing repeated error messages, other actions may be required.

Parameters: The *dev* argument is the device number (both major and minor device numbers).

The *string* argument is the string to be displayed on the console.

Syntax:

xxintr(vec_num)
int vec_num;

Description: The **xxintr** routine is called whenever the device issues an interrupt. Depending on the meaning of the interrupt, it may mark the current request as complete, start the next request, continue the current request, report a bad block, or retry a failed operation.

The routine examines the device status information, and determines whether the request was successful. The block buffer header is updated to reflect this. The interrupt routine checks to see if the device is idle and, if it is, starts it up before exiting.

Parameters: The value of *vec_num* is an integer that specifies the interrupt vector number of the device that originated the interrupt.

Note

Often a block device driver provides a character device driver interface, called *raw I/O*, so that the device can be accessed without going through the structuring and buffering imposed by the kernel's block device interface. For example, a program might wish to read magnetic tape records containing multiple block sizes, or read large portions of a disk directly. The use of the term raw I/O emphasizes the unstructured nature of the action. Adding the character device interface to a block device requires the **xxread**, **xxwrite** and **xxioctl** routines.

Syntax: **xxread(dev)**
dev_t dev;

Description: The **xxread** routine calls **physck**(K) and **physio**(K) with the appropriate arguments. This is the only action **xxread** performs.

Parameters: The value of *dev* specifies the major and minor device numbers.

Syntax: **xxwrite(dev)**
dev_t dev;

Description: The **xxwrite** routine calls **physck**(K) and **physio** with the appropriate arguments. This is the only action **xxwrite** performs.

Parameters: The value of *dev* specifies the major and minor device numbers.

Syntax: **xxioctl(dev, cmd, arg, mode)**
dev_t dev;
int cmd, mode;
caddr_t arg;

Description: The kernel calls the **xxioctl** routine when a user process makes an **ioctl**(S) system call for the specified device. This routine performs hardware-dependent functions such as parking the heads of a hard disk, setting a variable to indicate that the driver is to format the disk, or telling the driver to eject the media when the close routine is called.

Parameters: The value of *dev* specifies the major and minor device numbers.

The *cmd* argument specifies the command passed to the **ioctl**(S) system call.

The *arg* argument specifies the argument passed to the **ioctl** system call.

The *mode* argument specifies the flags set on the **open**(S) system call for the specified device.

Syntax:	**xxbreakup(bp)**
	struct buf *bp;

Description:	This routine is called to size the data in an I/O request into units that are usable by the type of controller being accessed. If the controller is a Direct Memory Access (DMA) controller, the size is generally set to 512-byte blocks. If the controller uses Programmed I/O (PIO), then the data is sized to an arbitrary value. The **xxbreakup** routine is called from **xxread** or **xxwrite** for performing raw I/O. When accessing a DMA controller, call **dma_breakup**(K); when using programmed I/O, call **pio_breakup**(K). If neither of these access methods are being used, **xxbreakup** is not needed. The two kernel breakup routines reformat the data in the I/O request and then call **xxstrategy** to perform raw I/O with the device.
Parameter:	The *bp* argument is a pointer to the buffer header.

Example Block Driver Code

The code examples presented here are for an intelligent controller that is attached to one or more disk drives. The controller can handle multiple sector transfers that cross track and cylinder boundaries.

Each segment of code is followed by general comments that describe the routines and explain key lines in the program.

IMPORTANT: These are not complete working drivers and should not be expected to be comprehensive. These example code fragments are meant to demonstrate implementations of individual items within a working driver. A complete working driver is listed in Appendix B, "Sample Block Driver."

For convenience, these code fragments are identified and referred to by line numbers.

Note also that the device drivers distributed in the */etc/conf/pack.d* directories may differ in design from those described in this document. When writing device drivers for the target system, you should follow these guidelines rather than examining the distributed drivers.

```
 /*
  * hd - prototype hard disk driver
  */

#include "sys/param.h"
#include "sys/buf.h"
#include "sys/iobuf.h"
#include "sys/dir.h"
#include "sys/conf.h"
#include "sys/user.h"

/* disk parameters */
#define NHD      4           /* number of drives */
#define NPARTS   8           /* # partitions/disk */
#define NCPD     600         /* # cylinders/disk */
#define NTPC     4           /* # tracks/cylinder */
#define NSPT     10          /* # sectors/track */
#define NBPS     512         /* # bytes/sector */
#define NSPB     (BSIZE/NBPS)       /* sectors/block */
#define NBPC     (NTPC*NSPT/NSPB)   /* blocks/cylinder */
#define NSPC     (NSPB/NBPC)        /* sectors/cylinder */
/* addresses of controller registers */
#define RBASE    0x00        /* base of all registers */
#define RCMD     (RBASE+0)   /* command register */
#define RSTAT    (RBASE+1)   /* status - nonzero = error */
#define RCYL     (RBASE+2)   /* target cylinder */
#define RTRK     (RBASE+3)   /* target track */
#define RSEC     (RBASE+4)   /* target sector */
#define RADDRL   (RBASE+5)   /* target mem addr lo 16 bits*/
#define RADDRH   (RBASE+6)   /* target mem addr hi 8 bits*/
#define RCNT     (RBASE+7)   /* number of sectors to xfer */

/* bits in RCMD register */
#define CREAD    0x01        /* start a read */
#define CWRITE   0x02        /* start a write */
#define CRESET   0x03        /* reset the controller */

/*
** minor number layout is 0000dppp
**  where d is the drive number and ppp is the partition
*/
#define drive(d)      ( minor(d) >> 3)
#define part(d)       ( minor(d) & 0x07)

/* partition table */
struct partab {
    daddr_t len;     /* # of blocks in partition */
    int     cyloff;  /* starting cylinder of partition */
};

```

The code is defined as follows:

Line no.	Definition
13:	NHD defines the number of drives to which the controller can be attached.
14:	NPARTS defines the number of partitions that can be configured on a single drive.
15-20:	Each disk drive attached to the controller has NCPD cylinders; each cylinder has NTPC tracks; and each track has NSPT sectors. The sectors are NBPS bytes long and each cylinder has NBPC blocks.
23-31:	The controller registers occupy a region of contiguous address space starting at RBASE and running through RBASE+7. To make the controller perform some action, the registers that describe the transfer (RCYL, RTRK, RSEC, RADDRL, RADDRH, RCNT) are set to the appropriate values.
34-36:	The bit representing the desired action is written into the RCMD register.
42-43:	The **drive** and **part** macros split out the two parts of the minor number. Bits 0 through 2 represent the partition on the disk, and the remaining bits specify the drive number. Thus, the minor number for drive 1, partition 2 would be 10 decimal.
46-50:	Large disks typically are broken into several partitions of a more manageable size. The structure that specifies the size of the partitions specifies the length of the partition in blocks, and the location of the starting cylinder of the partition.

hd_sizes: Lines 51 to 74

The following source code defines the partitions used for the device driver. The partition divides the disk into four separate areas.

```
51    int hdread(), hdwrite(), hdintr(), hdstrategy();
52
53    struct partab hd_sizes[8] = {
54        NCPD*NBPC,      0,             /* whole disk */
55        ROOTSZ*NBPC,    0,             /* root area */
56        SWAPSZ*NBPC,    ROOTSZ,  /* swap area */
57        USERSZ*NBPC,    USROFS,  /* user area */
58        0,      0,               /* spare */
59        0,      0,               /* spare */
60        0,      0,               /* spare */
61        0,      0,               /* spare */
62    };
63
64    struct  iobuf   hdtab;             /* start of request queue */
65    struct  buf     rhdbuf;            /* header for raw I/O */
66    /*
67    **    Strategy Routine:
68    **    Arguments:
69    **      Pointer to buffer structure
70    **    Function:
71    **      Check validity of request
72    **      Queue the request
73    **      Start up the device if idle
74    */
```

The code is defined as follows:

Line no.	**Definition**
54-57:	This driver splits a disk into up to eight partitions, but at present only four are used. The first partition covers the whole disk. The remaining three split the disk three ways, one partition for each of *root*, *swap*, and *user* areas.
64:	The buffer headers representing requests for this driver are linked into a queue, with *hdtab* forming the head of the queue. (*hdtab* is an example of the **xxtab** structure described previously in this chapter. *hdtab* is an instance of the *iobuf* structure described in *sys/iobuf.h*.) In addition, information regarding the state of the driver is kept in *hdtab*.

65: Each block driver that wants to allow raw I/O allocates one buffer header for this purpose.

hdstrategy: Lines 75 to 105

The **hdstrategy** routine is called by the kernel to queue a request for I/O. The single argument is a pointer to the buffer header which contains all of the data relevant to the request. The **xxstrategy** routine is responsible for validating the request, and linking it into the queue of outstanding requests.

```
75    int hdstrategy(bp)
76    register struct buf *bp;
77    {
78        register int dr, pa;    /* drive and partition numbers */
79        daddr_t sz, bn;
80        int x;
81        dr = drive(bp->b_dev);
82        pa = part(bp->b_dev);
83        bn = bp->b_blkno * NSPB;
84        sz = (bp -> b_bcount + BMASK) >> BSHIFT;
85        if ( dr < NHD && pa < NPARTS && bn >= 0
                 && bn < hd_sizes[pa].len &&
                ((bn + sz < hd_sizes[pa].len) ||
                 (bp->b_flags & B_READ)))
86        {
87            if ( bn + sz > hd_sizes[pa].len ) {
88                sz = (hd_sizes[pa].len - bn) * NBPS;
89                bp->b_resid = bp->b_bcount - (unsigned) sz;
90                bp->b_bcount = (unsigned) sz;
91            }
92        } else {
93            bp->b_flags |= B_ERROR;
94            iodone(bp);
95            return;
96        }
97        bp->b_cylin = (bp->b_blkno / NBPC) + hd_sizes[pa].cyloff;
98          bp->b_sector = (daddr_t) bp->b_cylin & 0xFFFF;
99        x = splbuf();
100       disksort(&hdtab, bp)
101       if (bp->b_active = = NULL)
102           hdstart();
103       splx(x);
104   }
105
```

The code is defined as follows:

Line no.	Definition
81-84:	First, compute various useful numbers that will be used repeatedly during the validation process.
85-96:	If the request is for a nonexistent drive or a nonexistent partition, if it lies completely outside the specified or is a write, and ends outside the partition, the B_ERROR bit in the *b_flags* field of the header is set to indicate that the request has failed. The request is then marked ''complete.'' This is done by calling **iodone**(K) with the pointer to the header as an argument. If the request is a read, and ends outside the partition, it is truncated to lie completely within the partition.
97 and 98:	Provide a field for **disksort**(K) to sort the I/O request so as to optimize disk head movement. In the case of these lines of code, the *b_cylin* and *b_sector* fields while not equal conceptually, can still be equated for the purpose of providing a basis for sorting as both express a linear grouping of disk information. Therefore, *b_cylin* is computed and then *b_sector* is set to the value of *b_cylin*. Because *b_cylin* is an unsigned short integer and *b_sector* is a signed long integer, care must be taken to ensure that sign extension does not occur. For future use, however, use of *b_cylin* should be curtailed as it is provided in this release of System V as a compatibility feature.
99:	Block interrupts to prevent the interrupt routine from changing the queue of outstanding requests.
100:	Sort the request into the queue by passing it and the head of the queue to the **disksort** routine.
101:	If the controller is not already active, start it up.
102:	Re-enable interrupts and return to the user process.

hdstart: Lines 106 to 139

The **hdstart** routine calculates the physical address on the disk, and starts the transfer.

```
106   /*
107    *      Startup Routine:
108    *      Arguments:
109    *        None
110    *      Function:
111    *        Compute device-dependent parameters
112    *        Start up device
113    *        Indicate request to I/O monitor routines
114    */
115   hdstart()
116   {
117       register struct buf *bp;       /* buffer pointer */
118       register unsigned sec;
119
120       if ((bp = hdtab.b_actf) = = NULL) {
121           hdtab.b_active = 0;
122           return;
123       }
124       hdtab.b_active = 1;
125
126       sec = ((unsigned)bp->blkno * NSPB);
127       outb(RCYL, sec / NSPC);            /* cylinder    */
128       sec %= NSPC;
129       outb(RTRK, sec / NSPT);            /* track       */
130       outb(RSEC, sec % NSPT);            /* sector      */
131       outb(RCNT, bp->b_count / NBPS);    /* count       */
132       outb(RDRV, drive(bp->b_dev));      /* drive       */
133       outb(RADDRL, paddr(bp) & 0xFFFF);  /* mem addr lo */
134       outb(RADDRH, paddr(bp) >> 16);     /* mem addr hi */
135       if ( bp->b_flags & B_READ )
136           outb(RCMD, CREAD);
137       else
138           outb(RCMD, CWRITE);
139   }
```

The code is defined as follows:

Line no.	**Definition**
120-122:	If there are no active requests, mark the state of the driver as idle, and return.
124:	Mark the state of the driver as active.
126-130:	Calculate the starting cylinder, track, and sector of the request, and load the controller registers with these values.
132-134:	Load the controller with the drive number, and the memory address of the data to be transferred.
135-139:	If the request is a read request, issue a read command; otherwise, issue a write command.

hdintr: Lines 140 to 174

The **hdintr** is called by the kernel whenever the controller issues an interrupt.

```
140
141    /*
142     *      Interrupt routine:
143     *        Check completion status
144     *        Indicate completion to I/O monitor routines
145     *        Log errors
146     *        Restart (on error) or start next request
147     */
148    hdintr()
149    {
150        register struct buf *bp;
151
152        if (hdtab.b_active = = 0)
153            return;
154
155        bp = hdtab.b_actf;
156
157        if ( inb(RSTAT) != 0 )
158            outb(RCMD, CRESET);
159            if (++hdtab.b_errcnt <= ERRLIM) {
160                hdstart();
161                return;
162            }
163            bp->b_flags |= B_ERROR;
164            deverr(&hdtab, bp, inb(RSTAT), 0);
165        }
166        /*
167         *      Flag current request complete, start next one
168         */
169        hdtab.b_errcnt = 0;
170        hdtab.b_actf = bp->av_forw;
171        bp->b_resid = 0;
172        iodone(bp);
173        hdstart();
174    }
```

The code is defined as follows:

Line no.	**Definition**
152-153:	If an unexpected call occurs, just return.
155:	Get a pointer to the first buffer header in the chain; this is the request that is currently being serviced.

157-165: If the controller indicates an error, and the operation hasn't been retried ERRLIM times, try it again. If it has been retried ERRLIM times, assume it is a hard error, mark the request as failed, and call **deverr**(K) to print a console message about the failure.

169-174: Mark this request complete, take it out of the request queue, and call **hdstart** to start on the next request.

hdread: Lines 175 to 195

The **hdread** routine is called by the kernel when a process requests raw read on the device. All it has to do is call **physio**, passing the name of the **xxstrategy** routine, a pointer to the raw buffer header, the device number, and a flag indicating a read request. The **physio** routine does all the preliminary work, and queues the request by calling the device **xxstrategy** routine.

```
175   /*  raw read routine:
176    *    This routine calls physio(K) which computes
177    *    and validates a physical address from the
178    *    current logical address.
179    *
180    *      Arguments
181    *        Device number
182    *      Functions:
183    *        Call physio to do the raw (physical) I/O
184    *        The arguments to physio are:
185    *          pointer to the xxstrategy routine
186    *          buffer for raw I/O
187    *          device number
188    *          read/write flag
189    */
190   hdread(dev)
191   dev_t dev;
192   {
193
194       physio(hdstrategy, &rhdbuf, dev, B_READ);
195   }
```

hdwrite: Lines 196 to 208

The **hdwrite** routine is called by the kernel when a process requests a raw write on the device. Its responsibilities and actions are the same as **hdread**, except that it passes a flag indicating a write request.

```
196
197    /*
198     *        Raw write routine:
199     *        Arguments(to hdwrite):
200     *          Full device number
201     *        Functions:
202     *          Call physio which does actual raw (physical) I/O
203     */
204    hdwrite(dev)
205    dev_t dev;
206    {
207        physio(hdstrategy, &rhdbuf, dev, B_WRITE);
208    }
```

Chapter 4

Character Device Drivers

Introduction to Character Devices 4-1

- Character Device Driver Routines 4-2
- Character Device Driver Interrupt Routine 4-8
- Character List and Character Block Architecture 4-8
- Terminal Device Drivers 4-10
- Other Character Devices 4-12

Example Character Driver Code 4-14

Code Fragments from a Line Printer Driver 4-15

Code Fragments From a Terminal Driver 4-21

Introduction to Character Devices

This chapter describes character device drivers, and provides sample character driver code examples. Other chapters in this book describe character drivers for specific applications. Once you have studied the concepts described in this chapter and examined the sample code at the end of the chapter, refer to the following chapters for information about other types of character drivers:

- Chapter 5 — Video Device Drivers
- Chapter 6 — Compiling and Linking Drivers
- Chapter 8 — Line Disciplines
- Chapter 9 — STREAMS

Character devices conform to the file model. Their data consists of a stream of bytes delimited only by the beginning and end of file. The operating system provides programs with direct access to devices through the special device files. For more information on special device files, see the section in Chapter 2 entitled, ''Special Device Files.''

Most character device drivers in the kernel are designed around the special requirements of terminal devices. There are facilities provided for programming functions on input and output (such as character erase, line kill, and tab functions), and for setting line options such as speed. Other character-oriented devices such as line printers use the same program interface as terminals, but with a different driver.

Character device drivers use ''*clists*'' for transferring relatively small amounts of data between the driver and the user program. A *clist* is an instance of *struct clist* defined in *sys/tty.h*. For more information about *clists*, see the section in this chapter entitled, ''Character List and Character Block Architecture.''

Character Device Driver Routines

The following table lists the driver routines that can be used in a character driver:

Routine	Purpose
xxopen	Start access to a character device
xxclose	End access to a character device
xxread	Transfer data from internal buffers to user space
xxwrite	Transfer data from user space to internal buffers
xxioctl	Perform I/O control commands
xxinit	Initialize the device when is booted
xxhalt	Executed when the computer is shut down
xxintr	Executed when an interrupt occurs
xxstart	Interact directly with the device
xxpoll	Executed on each clock tick
xxproc	Perform device-dependent line discipline I/O

The **xxproc** routine is described in Chapter 8, "Line Disciplines." Refer to the "Kernel Routines" section in Chapter 2 for information on the kernel routines used in a character driver.

The task-time portion of the character device driver is called when a user process requests a data transfer to or from a device under the control of the driver. The system determines which device is being called by reading the major device number of the device that the user wishes to use for I/O. The driver's job is to take the user process' requests, check the parameters supplied, and set up the necessary information to enable the device interrupt or poll routine to perform the I/O.

In the case of a write to a slow device (that is, one using *clists*), the driver copies the data from the user space into the output *clist* for the device. In the case of direct I/O between the device and user memory (for example, magnetic tapes), the driver simply sets up the I/O request. The routines that provide the interface between the kernel and character device drivers are described as follows. **xx** is a prefix that refers to the device type.

The following section provides more information about each driver routine:

Syntax:	**xxinit( )**
Description:	The **xxinit** routine initializes the device when the kernel is first booted. If present, it is called if you place an "i" in the second column table defined in the kernel configuration file */etc/conf/cf.d/mdevice*. The **init** routine should display a message on the console indicating that the associated device has been initialized. (To ensure consistency in **xxinit** routine startup messages, use the **printcfg**(K) kernel routine for displaying the console message.) The **xxinit** routine cannot contain calls to **sleep**(K), or access any *user* structure fields. If **timeout**(K) is called, the actual timing does not commence until the clock is initialized for use. Refer to **Intro**(K) for information on which kernel routines can be used in an initialization routine.

Syntax: **xxopen(dev, flag, id)**
dev_t dev
int flag, id;

Description: The **xxopen** routine prepares the device for the I/O transfers and performs any error or protection checking. It is called each time the device is opened.

Parameters: The following values are passed to the driver when the driver routine is called by the kernel:

The value of *dev* specifies the major and minor device numbers.

The values for *flag* can be:

- FAPPEND — open device for appended writes
- FEXCL — open device for exclusive access
- FNDELAY — open device without delay
- FREAD — open device for reading

- FSTOPIO — prevent further I/O (initiated by a security feature)
- FSYNC — open for synchronous writes
- FWRITE — open device for writing

The values for *id* can be:

- OTYP_CHR — open device for character I/O
- OTYP_LYR — open device context layer for use with **layers**(C)
- OTYP_MNT — open device for file system mount
- OTYP_SWP — open as swap device

Syntax: **xxclose(dev, flag)**
dev_t dev;
int flag;

Description: The **xxclose** routine is responsible for any cleanup that may be required, such as disabling interrupts and clearing device registers. It is called on the last close on a device.

Parameters: The value of *dev* specifies the major and minor device numbers.

The *flag* argument is the *oflag* argument passed to the last **open** system call.

Syntax: **xxstart( )**

Description: This routine is used to interact directly with the device's I/O ports (registers) using **inb**(K) and **outb**(K). This routine checks whether the device is ready to accept or send data, and if so, initiates the transfer, usually by sending a control word to the I/O port. **xxstart** is often called by both task-time and interrupt-time portions of the driver. **xxstart** is generally not used by device drivers that control tty devices.

Syntax: **xxhalt( )**

Description: The **xxhalt** routine, if present, is called when the system is shut down. This routine should be used to set or clear device registers so that devices will be ready for initialization after a warm boot. Care should be taken that all hardware and hardware controllers are reset as they would be by a power cycle.

Syntax: **xxintr(vec_num)**
int vec_num;

Description: The kernel calls the **xxintr** routine when the device issues an interrupt. Since the interrupt typically indicates completion of a data transfer, the interrupt routine must determine the appropriate action: perhaps taking the received character and placing it in the input buffer, or removing the next character from the output buffer and starting the transmission. Refer to **Intro**(K) for information on which kernel routines can be used in an interrupt routine.

Parameters: The value of *vec_num* is an integer that specifies the interrupt vector number of the device that originated the interrupt.

Syntax: **xxread(dev)**
dev_t dev;

Description: The **xxread** routine is called when a user program makes a read system call. The **xxread** subroutine transfers data to the user's address space. Use **copy-out**, **db_read**, **passc**, **subyte**, **suword**, or **ttread** in **xxread** to transfer data to the user.

Parameters: The value of *dev* specifies the major and minor device numbers.

Syntax: **xxwrite(dev)**
dev_t dev;

Description: The **xxwrite** routine is called when a user program makes a write system call. This routine transfers data from the user's address space. Use **copyin**(K), **cpass**(K), **db_write**, **fubyte**, **fuword**, or **ttwrite** in **xxwrite** to transfer data from the user.

Parameters: The value of *dev* specifies the major and minor device numbers.

Syntax:	**xxpoll(ps)**
	int ps;

Description:	The **xxpoll** routine, if present, is called by the system clock at **spl6**(K) during every clock tick. It is useful for repriming devices that constantly lose interrupts.
Parameters:	The value of *ps* is an integer that indicates the previous process' priority when it was interrupted by the system clock. The macro USERMODE(*ps*) (*ps*), defined in *sys/param.h*, can be used to determine if the interrupted process was executing in user mode.
Warnings	Only call **xxpoll** if your interrupt priority level is set at **spl6**(K) or higher. Otherwise, you will miss the interrupts at **spl6** described above.

Syntax:	**xxproc(tp, cmd)**
	struct tty *tp;
	int cmd;

Description:	This routine provides hardware-dependent code in a line discipline. Refer to "xxproc Routine" in Chapter 8, "Line Disciplines" for a description of this routine.
Parameters:	The *tp* argument is a pointer to an instance of the *tty* structure for a device. The *cmd* argument specifies the process to be performed.

Syntax:	**xxioctl(dev, cmd, arg, mode)**
	dev_t dev;
	int cmd, mode;
	caddr_t arg;

Description:	The kernel calls the **xxioctl** routine when a user process makes an **ioctl**(S) system call for the specified device. This routine performs hardware-dependent functions, such as setting the data rate on a character device.

Parameters: The value of *dev* specifies the major and minor device numbers.

The value of *cmd* is an integer that specifies the command passed to the system call.

arg specifies the argument passed to the system call.

The *mode* argument specifies the flags passed on the **open**(S) system call for the device.

Character Device Driver Interrupt Routine

The device interrupt routine is entered whenever a device associated with the driver raises an interrupt. Note that in general one driver may control several devices, but that all interrupts are vectored through a single routine entry point, usually called **xxintr**, where *xx* is a prefix that refers to the device type. It is the driver's responsibility to decide which device caused the interrupt.

When a device raises an interrupt, it makes available some status information to indicate the reason for the interrupt. The driver interrupt routine decodes this information. If it indicates a transfer just completed, the **wakeup**(K) routine alerts any process waiting for the transfer to complete. It then makes a check to see if the device is idle and, if so, looks for more work to start up. Thus, in the case of output to a terminal, the interrupt routine looks for more work in the *clist*s each time a transfer completes. Refer to the interrupt routines shown in the driver fragments at the end of this chapter.

Character List and Character Block Architecture

The character list (*clist*) structure provides a general character buffering system for use by character device drivers. The mechanism is designed for buffering small amounts of data from relatively slow devices, particularly terminals.

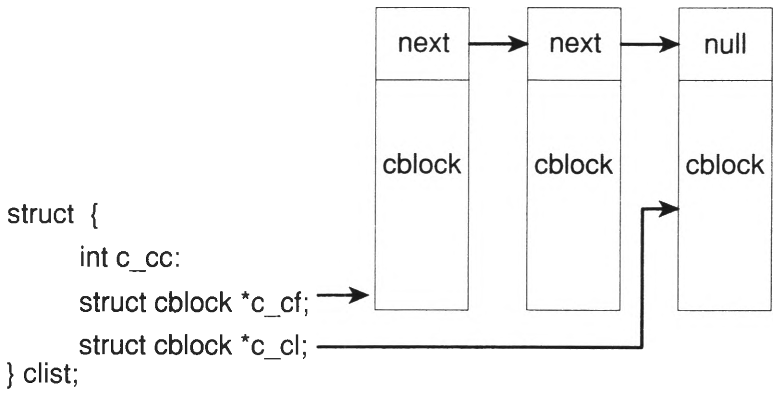
The kernel has a pool of character blocks called *cblocks*. Each *cblock* contains a link to the next *cblock* and an array of characters. A *clist* is a header to a linked-list queue of *cblock*s.

The **getc**(K) and **putc**(K) routines put characters into and remove characters from a *clist*. Drivers using *clist*s can use these routines. Note that the routines are not the same as the standard I/O library routines of the same name.

The static buffer header for each *clist* contains three fields:

- a count of the number of characters in the list (*c_cc*)
- a pointer to the first *cblock* in the list (*c_cf*)
- a pointer to the last *cblock* in the list (*c_cl*)

The *clist* buffers form a single linked list as shown in the following diagram:



A protocol is defined for use with the *clists* to prevent a particular process or driver from consuming all available resources. Two constants for the *clist* high- and low-water marks are defined in the file named *sys/tty.h*. A process can issue write requests until the corresponding *clist* hits the high-water mark. The process is then suspended and I/O performed. When the list reaches the low-water mark, the process is awakened. Read requests use a similar protocol.

Terminal Device Drivers

Terminal device drivers use *clists* extensively. For each terminal line (each minor device number), the driver declares **static** *clist* headers for three *clists*. These *clists* are the:

- Raw input queue, **t_rawq**
- Canonical queue, **t_canq**
- Output queue, **t_outq**

When a process writes data to a terminal device, the task-time portion of the driver puts the data into the output queue, and the interrupt routine transfers it from the queue to the device.

When a process requests a read of data from the terminal, the situation is slightly more complicated. This is because the operating system provides for some processing of characters on input, at the option of the requesting process. For example, in normal input the <Bksp> key is interpreted as ''delete the last character input,'' and the line-kill character means ''delete the whole current line.'' Certain special characters, such as <Bksp>, have to be treated in context; that is, they depend upon surrounding characters. To handle this, drivers use two queues for incoming data.

These two queues are the raw queue and the canonical queue. Data received by the interrupt routine is placed in the raw queue with no data processing. At task time, the driver decides how much processing to do. The user process has the option of requesting *raw* input, in which it receives data directly from the raw queue. *Cooked* (the opposite of raw) input refers to the input after processing for ERASE, line kill, DELETE, and other special treatment. In this case, a task-time routine, **canon**(K), transfers data from the raw queue to the canonical queue. This performs ⟨BKSP⟩ and line-kill functions, according to the options set by the process using the **ioctl**(S) system call.

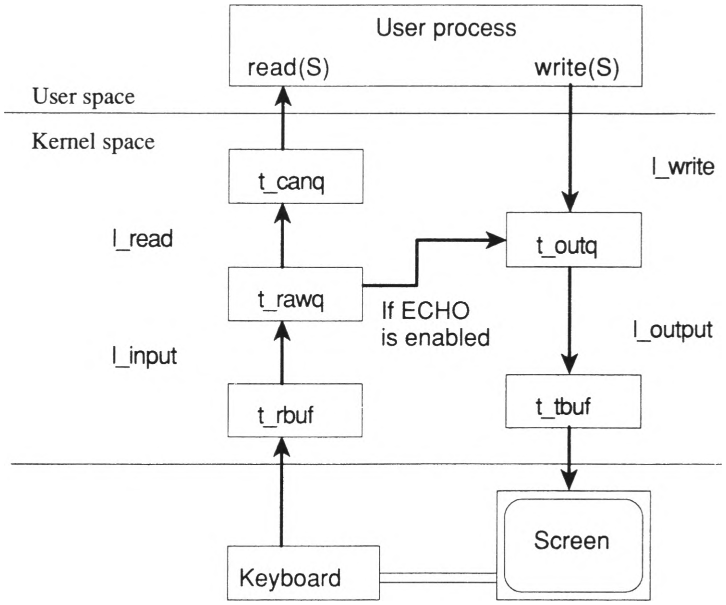
In System V, the direct *clist* processing for tty device drivers is normally handled by the specific line discipline. (Line disciplines are described in detail in Chapter 8, "Line Disciplines.") A line discipline is a set of kernel and driver routines that process data received from a tty device, such as a terminal. The processing generally follows the guidelines described on the **termio**(M) manual page, but can be customized as needed.

The only processing that the device driver needs to perform is interrupt level control. The device driver provides interrupt level control by emptying and filling structures called character-control blocks (*ccblock*). Each *tty* structure has a *ccblock* for transmitter (*t_tbuf*) control and a *ccblock* for receiver (*t_rbuf*) control. The *ccblock* structure has the following format:

```
struct ccblock {
      caddr_t    c_ptr;           /*buffer address*/
      ushort_t   c_count;         /*character count*/
      ushort_t   c_size;          /*buffer size*/
};
```

At receiver interrupt time, the driver fills a receiver *ccblock* with characters, decrements the character count, and calls the line-discipline routine **l_input**. At transmitter interrupt time, the driver calls **xxproc** and the line-discipline routine, **l_output**, to get a transmitter *ccblock*, and then outputs as many characters as possible.

The basic flow of data through the system during terminal I/O is shown in the following diagram:



There are two slight complications to the scheme presented in the preceding diagram. These are *output character expansion*, and *input character echo*.

Output expansion occurs for a few special characters. In cooked mode, tabs may be expanded into spaces, and the NEWLINE character is mapped into RETURN plus LINEFEED. There is a facility for producing escape sequences for upper case terminals, and delay periods for certain characters on slow terminals. Note that all of these are simple expansions, or mapping single characters, and so do not require a second list, as is the case for input. Instead, all the expansion is performed by the **xxproc** routine before placing the characters in the output *clist*.

Character echo is an option required by most user processes. With this option, all input characters are immediately echoed to the output stream without waiting for the user process to be scheduled. Character expansion is performed for echoed characters, as it is for regular output. Character echo takes place at interrupt time, so that a user typing at a terminal gets fast echo, regardless of whether his program is in memory and running, or is swapped out to a disk.

Other Character Devices

The following are common character devices:

- the console
- terminals
- line printers
- magnetic tape drives

The system console driver has two sections: a generic device-independent section that can talk to any of the many varieties of video adapters, and a device-specific section that a driver writer provides with information about a single video adapter. See Chapter 5, "Video Adapter Device Drivers," for more information.

Terminals receive a lot of special attention in the operating system. Line printers and magnetic tape drives tend to use existing kernel facilities with little special handling.

Line Printers

Line printers are relatively slow, character-oriented devices. The drivers use the *clist* mechanism for buffering data. However a line printer driver is generally simpler than a terminal driver because there is less processing of output characters to do, and there is no input.

Magnetic Tape Drives

Magnetic tape is a special case. The data is arranged on the physical medium in blocks, as on a disk. However, it is almost always accessed serially. Furthermore, there is generally only one program accessing a tape drive at a time. Thus, the management scheme of the kernel buffer is not applicable to tapes. Nor is the *clist* mechanism applicable, because of the large amount of data involved.

Usually tape drivers provide two interfaces: a *block* and a *character* interface. The character interface is used for raw, or physical, I/O directly between the device and the user process' address space. The block interface makes use of the kernel buffer pool and buffer manipulation routines to store data in transit between device and process. For more information on providing the facility for raw I/O, see the section in Chapter 3, ''Character Interface to Block Devices.''

Example Character Driver Code

This section provides code fragments from example drivers for a line printer and a terminal. Each segment of code is followed by general comments that describe the routines and explain key lines in the program.

IMPORTANT: These are not complete working drivers and should not be expected to be comprehensive. These example code fragments are meant to demonstrate implementations of individual items within a working driver.

For convenience, these code fragments are identified and referred to by line numbers.

Note also that the device drivers distributed in the */etc/conf/pack.d* directories may differ in design from those described in this document. When writing device drivers for the target system, you should follow these guidelines rather than examining the distributed drivers.

Code Fragments from a Line Printer Driver

The examples presented here are part of a driver providing a single, parallel interface to a printer. The driver transfers characters, one at a time, buffering the output from the user process through the use of character blocks (*cblocks*).

```
/*
** lp - prototype line printer driver
*/
#include "sys/param.h"
#include "sys/dir.h"
#include "sys/a.out.h"
#include "sys/user.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/conf.h"


#define LPPRI   PZERO+5
#define LOWAT   50
#define HIWAT   150

/* register definitions */

#define RBASE   0x00            /* base address of registers */
#define RDATA   (RBASE + 0)     /* place character here */
#define RSTATUS (RBASE + 1)     /* non zero means busy */
#define RCNTRL  (RBASE + 2)     /* write control here */

/* control definitions */
#define CINIT          0x01   /* initialize the interface */
#define CIENABL        0x02   /* interrupt enable */

/* flags definitions */
#define FIRST      0x01
#define ASLEEP           0x02
#define ACTIVE           0x04

struct clist lp_queue;
int lp_flags = 0;

int lpopen(), lpclose(), lpwrite(), lpintr();


```

The code is defined as follows:

Line no.	**Definition**
13:	LPPRI is the priority at which a process sleeps when it needs to stop. Since the priority is greater than PZERO, a signal sent to the suspended process will awaken it.
14:	LOWAT is the minimum number of characters in the buffer. If there are fewer than LOWAT characters in the buffer, a process that was suspended (because the buffer was full) can be restarted.
15:	HIWAT is the maximum number of characters in the queue. If a process fills the buffer up to this point, it will be suspended by means of **sleep** until the buffer has drained below LOWAT.
19-22:	The device registers in this interface occupy a contiguous block of address, starting at RBASE, and running through RBASE+2. The data to be printed is placed in RDATA, one character at a time. Printer status can be read from RSTATUS, and the interface can be configured by writing into RCNTRL.
29-31:	The flags defined in these lines are kept in the *lp_flags* variable. FIRST is set if the interface has been initialized. ASLEEP is set if a process is asleep, waiting for the buffer to drain below LOWAT. ACTIVE is set if the printer is active.
33:	*lp_queue* is the head of the linked list of cblocks that forms the output buffer.
34:	*lp_flags* is the variable in which the aforementioned flags are kept.

lpopen: Lines 39 to 48

The **lpopen** routine is called when some process makes an **open**(S) system call on the special file that represents this driver. Its single argument, *dev*, represents the major and minor device numbers. Since this driver supports only one device, the device number is ignored.

```
39
40    lpopen(dev)
41    dev_t dev;
42    {
43            if ( (lp_flags & FIRST) = = 0 ) {
44                    lp_flags |= FIRST;
45                    outb(RCNTRL, CRESET);
46            }
47            outb(RCNTRL, CIENABL);
48    }
```

The code is defined as follows:

Line no.	**Definition**
43-45:	If this is the first time (since the system was booted) that the device has been touched, the interface is initialized by setting the CRESET bit in the control register.
47:	Interrupts from this device are enabled by setting the CIENABL bit in the control register.

lpclose: Lines 49 to 53

The **lpclose** routine is called on the last close of the device; that is, when the current **close**(S) system call results in zero processes referencing the device. No action is taken.

```
49
50    lpclose(dev)
51    dev_t dev;
52    {
53    }
```

lpwrite: Lines 54 to 73

The **lpwrite** routine is called to move the data from the user process to the output buffer.

```
54    lpwrite(dev)
55    dev_t dev;
56    {
57            register int c;
58            int x;
59
60            while ( (c = cpass()) >= 0 ) {
61                    x = splcli();
62                    while ( lp_queue.c_cc > HIWAT ) {
63                            lpstart();
64                            lp_flags |= ASLEEP;
65                            sleep(&lp_queue, LPPRI);
66                    }
67                    putc(c, &lp_queue);
68                    splx(x);
69            }
70            x = splcli();
71            lpstart();
72            splx(x);
73    }
```

The code is defined as follows:

Line no.	**Definition**
60:	While there are still characters to be transferred, do what follows.
61-68:	Raise the processor priority so the interrupt routine cannot change the buffer. If the buffer is full, make sure the printer is running, note that the process is waiting, and put it to sleep. When the process wakes up, check to make sure the buffer has enough space, then go back to the old priority and put the character in the buffer.
70-71:	Make sure the printer is running by locking out interrupts and calling **lpstart**.

lpstart: Lines 74 to 81

The **lpstart** routine ensures that the printer is running. It is called twice from **lpwrite** and serves simply to avoid duplicate code.

```
 74
 75    lpstart()
 76    {
 77        if ( lp_flags & ACTIVE )
 78            return; /* interrupt chain is keeping printer going */
 79        lp_flags |= ACTIVE;
 80        lpintr(0);
 81    }
```

The code is defined as follows:

Line no.	Definition
77-80:	If the printer is running, just return; otherwise, set ACTIVE, and call **lpintr** to start the transfer of characters.

lpintr: Lines 82 to 105

The **lpintr** routine is called from two places: **lpstart**, and from the kernel interrupt-handling sequence when a device interrupt occurs.

```
 82
 83
 84    lpintr(vec)
 85    int vec;
 86    {
 87        int tmp;
 88
 89        if ( (lp_flags & ACTIVE) = = 0 )
 90            return;            /* ignore spurious interrupt */
 91
 92        /* pass chars until busy */
 93        while ( inb(RSTATUS) = = 0 && (tmp = getc(&lp_queue)) >= 0)
 94            outb(RDATA, tmp);
 95
 96        /* wakeup the writer if necessary */
 97        if ( lp_queue.c_cc < LOWAT && lp_flags & ASLEEP ) {
 98            lp_flags &= ~ASLEEP;
 99            wakeup(&lp_queue);
100    }
101
102         /* wakeup writer if waiting for drain */
103         if ( lp_queue.c_cc <= 0 )
104             lp_flags &= ~ACTIVE;
105     }
```

The code is defined as follows:

Line no.	Definition
89-90:	If **lpintr** is called unexpectedly, or the driver does not have anything to do, it just returns.
93-94:	While the printer indicates it can take more characters and the driver has characters to give it, the characters come from the buffer through **getc**(K) and pass to the interface by writing to the data register.
97-99:	If the buffer has fewer than LOWAT characters in it and some process is asleep waiting for room, wake it up.
103-104:	If the queue is empty, turn off the ACTIVE flag. Note that the interrupt that completes the transfer and empties the buffer is in some sense ''spurious,'' since it will occur with the ACTIVE flag reset.

Code Fragments From a Terminal Driver

The following examples are from a driver that supports one serial terminal on a hypothetical universal asynchronous receiver-transmitter (UART) type interface. Note that this is not the entire driver and that the situation is hypothetical.

```
/*
 * td - terminal device driver
 */
#include "sys/param.h"
#include "sys/dir.h"
#include "sys/user.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/conf.h"


/* registers */
#define RRDATA    0x01      /* received data */
#define RTDATA    0x02      /* transmitted data */
#define RSTATUS   0x03      /* status */
#define RCNTRL    0x04      /* control */
#define RIENABL   0x05      /* interrupt enable */
#define RSPEED    0x06      /* data rate */
#define RIIR      0x07      /* interrupt identification */

/* status register bits */
#define SRRDY     0x01      /* received data ready */
#define STRDY     0x02      /* transmitter ready */
#define SOERR     0x04      /* received data overrun */
#define SPERR     0x08      /* received data parity error */
#define SFERR     0x10      /* received data framing error */
#define SDSR      0x20      /* status of dsr (cd)*/
#define SCTS      0x40      /* status of clear to send */

/* control register */
#define CBITS5    0x00      /* five bit chars */
#define CBITS6    0x01      /* six bit chars */
#define CBITS7    0x02      /* seven bit chars */
#define CBITS8    0x03      /* eight bit chars */
#define CDTR      0x04      /* data terminal ready */
#define CRTS      0x08      /* request to send */
#define CSTOP2    0x10      /* two stop bits */
#define CPARITY   0x20      /* parity on */
#define CEVEN     0x40      /* even parity otherwise odd */
#define CBREAK    0x80      /* set xmitter to space */

```

```
42      /* interrupt enable */
43      #define EXMIT     0x01      /* transmitter ready */
44      #define ERECV     0x02      /* receiver ready */
45      #define EMS       0x04      /* modem status change */
46
47      /* interrupt ident */
48      #define IRECV     0x01
49      #define IXMIT     0x02
50      #define IMS       0x04
51
52      #define NIDEVS    2
53      #define VECT0     3
54      #define VECT1     5
```

The code is defined as follows:

Line no.	**Definition**
13-19:	The interface for each line consists of seven registers. The values that would be defined here represent offsets from the base address, which is defined in line 101. The base address differs for each line. The data to be transmitted is placed one character at a time into the RTDATA register. Likewise, the received data is read one character at a time from the RRDATA register. You can determine the status of the UART by examining the contents of the RSTATUS register. Then you can adjust the UART configuration by changing the contents of the RCNTRL register. Interrupts are enabled or disabled by setting the bits in the RIENABL register. The data rate is set by changing the contents of the RSPEED register. Interrupts are identified by setting the bits in the RIIR register.
31-40:	The two low-order bits of the *control register* determine the length of the character sent. The next two bits control the data-terminal-ready and request-to-send lines of the interface. The next bit controls the number of stop bits, the next controls whether parity is generated, and the next controls whether generated parity is even or odd. Finally, the most significant bit, if it is set, forces the transmitter to continuous spacing.
43-45:	The three low-order bits of the *interrupt enable* register control whether the device generates interrupts under certain conditions. If bit 0 is set, an interrupt is

generated every time the transmitter becomes ready for another character. If bit 1 is set, an interrupt is generated every time a character is received. If bit 2 is set, an interrupt is generated every time the data-set-ready line changes state.

48-50: After an interrupt, the value in the interrupt-identification register will contain one of three values, indicating the reason for the interrupt.

td_speeds: Lines 55 to 80

The array of integers, *td_speeds*, defines the data rates available to the device.

```
55     int tdopen(), tdclose(), tdread(), tdwrite(), tdioctl(),
       tdintr();
56
57
58
59     /* data rates */
60     int td_speeds[] = {
61             /* B0    */       0,
62             /* B50   */    2304,
63             /* B75   */    1536,
64             /* B110  */    1047,
65             /* B134  */     857,
66             /* B150  */     768,
67             /* B200  */       0,
68             /* B300  */     384,
69             /* B600  */     192,
70             /* B1200 */      96,
71             /* B1800 */      64,
72             /* B2400 */      48,
73             /* B4800 */      24,
74             /* B9600 */      12,
75             /* EXTA  */       6, /* 19.2k bps */
76             /* EXTB  */      58  /* 2000 bps  */
77         };
78
79     struct tty td_tty[NTDEVS];
80     int td_addr[NTDEVS] = { 0x00, 0x10 };
```

The code is defined as follows:

Line no.	**Definition**
59-77:	These lines define the values to be loaded into the RSPEED register in order to get various data rates.
79:	Each line must have a *tty* structure allocated for it.
80:	Here, the base addresses of the registers are defined for each line.

tdopen: Lines 81 to 123

The **tdopen** routine is called whenever a process makes an **open**(S) system call on the special file corresponding to this driver.

```
81
82
83      tdopen(dev, flag)
84      dev_t dev, flag;
85      {
86              register struct tty *tp;
87              int addr;
88              tdproc;
89              int x;
90              dev = minor(dev);
91              if ( UNMODEM(dev) >= NTDEVS ) {
92                      seterror(ENXIO);
93                      return;
94              }
95              tp = &td_tty[UNMODEM(dev)];
96              addr = td_addr[UNMODEM(dev)];
97              if( (tp->t_lflag & XCLUDE) && !suser ) {
98                      seterror(EBUSY);
99                      return;
100             }
101             if ((tp->t_state&(ISOPEN|WOPEN)) = = 0) {
102                     ttinit(tp);
103                     tp->t_proc = tdproc;
104                     tp->t_oflag = OPOST|ONLCR;
105                     tp->t_iflag = ICRNL|ISTRIP|IXON;
106                     tp->t_lflag = ECHO|ICANON|ISIG|ECHOE|ECHOK;
107                     tdparam(dev);
108             }
109          x = splcli();
110          if ( ISMODEM(dev) ||
111             tp->t_cflag & CLOCAL ||
112             tdmodem(dev, TURNON))
113                  tp->t_state |= CARR_ON;
114          else
115                  tp->t_state &= ~CARR_ON;
116          if (!(flag&FNDELAY))
117                  while ((tp->t_state&CARR_ON)==0) {
118                              tp->t_state |= WOPEN;
119                          sleep((caddr_t)&tp->t_canq, TTIPRI);
120                  }
121          (*linesw[tp->t_line].l_open)(tp);
122          splx(x);
123     }
```

The code is defined as follows:

Line no.	**Definition**
90:	convert the device number (contains both a major and a minor number) to just a minor number using the **minor**(K) kernel routine (described on the **major**(K) manual page).
91-93:	If the minor number indicates a device that does not exist, indicate the error and return.
97-99:	If the line is already open for exclusive use, and the current user is not the superuser, indicate the error and return.
101-107:	If the line is not already open, initialize the *tty* structure by means of a call to **ttinit**(K), set the value of the *proc* field in the *tty* structure, initialize the input and output mode flags, and configure the line by calling **tdparam**. Note that the flags are initialized so that the terminal will behave in a reasonable manner if used as the console in single-user mode.
109:	Defer interrupts so the interrupt routines cannot change the state while it is being examined.
110-115:	If the line is not using modem control, or if it is not turning on the data-terminal-ready and request-to-send signals (which results in carrier-detect being asserted by the remote device), indicate that the

	carrier signal is present on this line. Otherwise, indicate that there is no carrier signal.
116-119:	If the open is supposed to wait for the carrier, wait until the carrier is present.
121:	Call the **l_open** routine indirectly through the *linesw* table. The line discipline switch table, *linesw*, is described in more detail in Chapter 8, "Line Disciplines." This completes the work required for the current line discipline to open a line.
122:	Allow further interrupts.

tdclose: Lines 124 to 136

The **tdclose** routine is called on the last close on a line.

```
124     tdclose(dev)
125     dev_t dev;
126     {
127             register struct tty *tp;
128             dev = minor(dev);    /* get minor device num */
129             tp = &td_tty[UNMODEM(dev)];
130             (*linesw[tp->t_line].l_close)(tp);
131             if (tp->t_cflag & HUPCL)
132                     tdmodem(dev, TURNOFF);
133             tp->t_lflag &= ~XCLUDE; /* turn off exclusive */
134             /*  use bit, and turn off interrupts */
135             outb(td_addr[UNMODEM(dev)] + RIENABL, 0);
136     }
```

The code is defined as follows:

Line no.	**Definition**
128:	Get the minor number from the device number (which contains both the major and minor device numbers).
130:	Call the **ttclose** routine through the *linesw* table to do the work required by the current line discipline.
131-132:	If the ‘‘hang up on last close’’ bit is set, drop the data-terminal-ready and request-to-send signals.

133:	Reset the exclusive-use bit.
135:	To prevent spurious interrupts, disable all interrupts for this line.

tdread and tdwrite: Lines 137 to 148

The **tdread** and **tdwrite** routines call the relevant routine by means of the *linesw* table. The called routine performs the appropriate action for the current line discipline.

```
137   tdread(dev)
138   dev_t dev;
139   {
140      dev = minor(dev);
141      (*linesw[tp->t_line].l_read)(&td_tty[UNMODEM(dev)]);
142   }
143   tdwrite(dev)
144   dev_t dev;
145   {
146      dev = minor(dev);
147      (*linesw[tp->t_line].l_write)(&td_tty[UNMODEM(dev)]);
148   }
```

tdparam: Lines 149 to 184

The **tdparam** routine configures the line to the mode specified in the appropriate *tty* structure.

```
149     tdparam(dev)
150     dev_t dev;
151     {
152             register int cflag;
153             register int addr;
154             register int temp, speed, x;
155             dev = minor(dev);

156             addr = td_addr[UNMODEM(dev)];
157             cflag = td_tty[UNMODEM(dev)].t_cflag;

158             /* if speed is B0, turn line off */
159             if ( (cflag & CBAUD) = = B0){
160               outb(addr + RCNTRL, inb(addr+RCNTRL) &
                        ~CDTR & ~CRTS);
161               return;
162             }
163
164             /* set up speed */
165             outb( addr + RSPEED, td_speeds[ cflag & CBAUD ]);
166
167             /* set up line control */
168             temp = (cflag & CSIZE) >> 4; /* length */
169             if ( cflag & CSTOPB )
170                     temp |= CSTOP2;
171             if ( cflag & PARENB ) {
172                     temp |= CPARITY;
173                     if ( (cflag & PARODD) = = 0)
174                             temp |= CEVEN;
175             }
176             temp |= CDTR | CRTS;
177             outb( addr + RCNTRL, temp );
178
179             /* setup interrupts */
180             temp = EXMIT;
181             if ( cflag & CREAD )
182                     temp |= ERECV;
183             outb(addr + RENABL, inb(RENABL) | temp);
184     }
```

The code is defined as follows:

Line no.	**Definition**
156-157:	Get the base address and flags for the referenced line.
159-161:	The speed B0 means ‘‘hang up the line.’’
165-184:	The remainder of the **tdparam** routine simply loads the device registers with the correct values.

tdmodem: Lines 185 to 203

The **tdmodem** routine controls the data-terminal-ready and request-to-send line signals. Its return value indicates whether data-set-ready signal (carrier detect) is present for the line.

```
185     tdmodem(dev, cmd)
186     dev_t dev;
187     int cmd;
188     {
189             register int addr;
190             dev = minor(dev);
191             addr = td_addr[UNMODEM(dev)];
192             switch(cmd){
193             case TURNON: /* enable modem interrupts,
                                set DTR & RTS true */
194               outb(addr + RENABL, inb(RENABL) | EMS);
195               outb(addr + RCNTRL, inb(RENABL)
                             | CDTR | CRTS );
196               break;
197             case TURNOFF: /* disable modem interrupts,
                                 reset DTR, RTS */
198               outb(addr + RENABL, inb(RENABL) &~EMS);
199               outb(addr + RCNTRL, inb(RENABL) & ~(CDTR|CRTS));
200               break;
201             }
202             return (inb(addr + RSTATUS) & SDSR);
203     }
```

The code is defined as follows:

Line no.	Definition
193-196:	If *cmd* was TURNON, turn on modem interrupts, and assert data-terminal-ready and request-to-send.
197-200:	If *cmd* was TURNOFF, disable modem interrupts, then drop data-terminal-ready and request-to-send.
202:	Return a zero value if there is no data-set-ready on this line; otherwise return a nonzero value.

tdintr: Lines 204 to 230

The **tdintr** routine determines which line caused the interrupt and the reason for the interrupt, and calls the appropriate routine to handle the interrupt.

```
204
205
206     tdintr(vec)
207     int vec;
208     {
209             register int iir, dev;
210
211             switch( vec ) {
212                     case VECT0:
213                             dev = 0;
214                             break;
215                     case VECT1:
216                             dev = 1;
217                             break;
218                     default:
219                             printf("tdint: wrong level
                                              interrupt (%x)\n",vec);
220                             return;
221                     }
222             while( (iir = inb(td_addr[dev]+RIIR)) != 0) {
223                     if( (iir & IXMIT) != 0 )
224                                        tdxint(dev);
225                                 if( (iir & IRECV) != 0 )
226                                        tdrint(dev);
227                                 if( (iir & IMS) != 0 )
228                                        tdmint(dev);
229                       }
230         }
```

The code is defined as follows:

Line no.	Definition
211-221:	Different lines will result in different interrupt vectors being passed as the **tdintr** routine's argument. Here, the minor number is determined from the interrupt vector that was passed to **tdintr**.
222-230:	While the interrupt-identification register indicates that there are more interrupts, call the appropriate routine. When the condition that caused the interrupt is resolved, the UART will reset the bit in the register by itself.

tdxint: Lines 231 to 251

The **tdxint** routine is called when a transmitter ready interrupt is received. It may issue a CSTOP character to indicate that the device on the other end must stop sending characters. It may issue a CSTART character to indicate that the device on the other end may resume sending characters, or it may call **tdproc** to send the next character in the queue.

```
231      tdxint(dev)
232      dev_t dev;
233      {
234          register struct tty *tp;
235          register int addr;
236          dev = minor(dev);
237          tp = &td_tty[UNMODEM(dev)];
238          addr = td_addr[UNMODEM(dev)];
239          if ( inb(addr + RSTATUS) & STRDY )
240          {
241                tp->t_state &= ~BUSY;
242                if (tp->t_state & TTXON) {
243                      outb(addr + RTDATA, CSTART);
244                      tp->t_state &= ~TTXON;
245                } else if (tp->t_state & TTXOFF) {
246                      outb(addr + RTDATA, CSTOP);
247                      tp->t_state &= ~TTXOFF;
248                } else
249                      tdproc(tp, T_OUTPUT);
250          }
251      }
```

The code is defined as follows:

Line no.	**Definition**
239-241:	If the transmitter is ready, disable the busy state.
242-244:	If the line is to be restarted, send a CSTART, and reset the state indicator.
245-247:	If the line is to be stopped, send a CSTOP, and reset the state indicator.
248-249:	Otherwise, call **tdproc** and ask it to send the next character in the queue.

tdrint: Lines 252 to 317

The **tdrint** routine is called when a receiver interrupt is received. All it has to do is pass the character, along with any errors, to the appropriate routine by means of the *linesw* table.

```
252  tdrint(dev)
253  dev_t dev;
254  {
255       register int c, status;
256       register int addr;
257       register struct tty *tp;
258       dev = minor(dev);
259       tp = &td_tty[UNMODEM(dev)];
260       addr = td_addr[UNMODEM(dev)];
261
262       /* get char and status */
263       c = inb( addr + RRDATA );
264       status = inb(addr + RLSR);
265
266       /*
267        *  Were there any errors on input?
268        */
269       if( status & SOERR )      /* overrun error */
270            c |= OVERRUN;
271       if( status & SPERR )      /* parity error */
272            c |= PERROR;
273       if( status & SFERR )      /* framing error */
274            c |= FRERROR;
275
276       if (tp->t_rbuf.c_ptr == NULL)
277            return;
278          flg = tp->t_iflag;
279          if (flg&IXON) {
280                  register int ctmp;
281                  ctmp = c & 0177;
282                  if( tp->t_state & TTSTOP ) {
283                          if (ctmp == CSTART || flg&IXANY)
284                                  (*tp->t_proc)(tp, T_RESUME);
285                  } else {
286                          if (ctmp == CSTOP)
287                                  (*tp->t_proc)(tp, T_SUSPEND);
288                  }
289                  if (ctmp == CSTART || ctmp == CSTOP)
290                          return;
291          }
292          if (c&PERROR && !(flg&INPCK))
293                  c &= ~PERROR;
294          if (c&(FRERROR|PERROR|OVERRUN)) {
295                  if ((c&0377) == 0) {
296                          if (flg&IGNBRK)
297                                  return;
298                          if (flg&BRKINT) {
299                                  (*linesw[tp->t_line].l_input)
300                                                  (tp, L_BREAK);
301                                  return;
302                          }
303                  } else {
304                          if (flg&IGNPAR)
305                                  return;
306                  }
307          } else {
308                  if (flg&ISTRIP)
309                          c &= 0177;
310                  else {
311                          c &= 0377;
312                  }
313          }
314          *tp->t_rbuf.c_ptr = c;
315          tp->t_rbuf.c_count--;
316          (*linesw[tp->t_line].l_input)(tp, L_BUF);
317  }
```

The code is defined as follows:

Line no.	**Definition**
262-264:	Get the character and status.
269-274:	If any errors were detected, set the appropriate bit in *c*.

279-291: This code determines whether the character is XON and, if output is stopped, it restarts it. If the character is XOFF, output is suspended.

292-313: Further error checking is then carried out and characters in error are discarded. The character is then placed in the queue.

314-315: This code stores the character into the received buffer and decrements the character count.

316: Finally, character and errors are passed to the **l_input** routine for the current line discipline.

tdmint: Lines 318 to 345

The **tdmint** routine is called whenever a modem interrupt is caught.

```
318  tdmint(dev)
319  dev_t dev;
320  {
321          register struct tty *tp;
322          register int addr,c;
323          dev = minor(dev);
324          tp = &td_tty[UNMODEM(dev)];
325          if ( tp->t_cflag & CLOCAL ) {
326                  return;
327          }
328          addr = td_addr[UNMODEM(dev)];
329
330          if (inb(addr + RSTATUS) & SDSR) {
331                  if ((tp->t_state & CARR_ON)= =0) {
332                          tp->t_state |= CARR_ON;
333                          wakeup(&tp->t_canq);
334                  }
335          } else {
336                  if (tp->t_state & CARR_ON) {
337                          if (tp->t_state & ISOPEN) {
338                                  signal(tp->t_pgrp, SIGHUP);
339                                  tdmodem(dev, TURNOFF);
340                                  ttyflush(tp, (FREAD|FWRITE));
341                          }
342                          tp->t_state &= ~CARR_ON;
343                  }
344          }
345  }
```

Code Fragments From a Terminal Driver

The code is defined as follows:

Line no.	Definition
325-326:	If there is no modem support for this line, just return.
330-333:	If a data-set-ready is present for this line, and it did not exist before, mark the line as having a carrier, and wake up any processes that are waiting for the carrier before their **tdopen** call can be completed.
335-344:	If no data-set-ready is present for this line, and one existed before, send a hangup signal to all of the processes associated with this line, call **tdmodem** to hang up the line, flush the output queue for this line by calling **ttyflush**(K), and mark the line as having no carrier.

tdioctl: Lines 346 to 355

The **tdioctl** routine is called when some process makes an **ioctl**(S) system call on a device associated with the driver. It just calls **ttiocom**, which returns a nonzero value if the hardware must be reconfigured.

```
346   tdioctl(dev, cmd, arg, mode)
347   dev_t dev;
348   int cmd;
349   caddr_t arg;
350   int mode;
351   {
352       dev = minor(dev);
353       if(ttiocom(&td_tty[UNMODEM(dev)],cmd,arg,mode))
354           tdparam(dev);
355   }
```

tdproc: Lines 356 to 433

The **tdproc** routine is called to effect some change on the output, such as emitting the next character in the queue, or halting or restarting the output.

```

tdproc(tp, cmd)
register struct tty *tp;
{
        register c;
        register int addr;

        extern ttrstrt;

        addr = td_addr[tp - td_tty];
        switch (cmd) {

        case T_TIME:
                tp->t_state &= ~TIMEOUT;
                outb(addr + RCNTRL,
                                 inb(addr + RCNTRL) & ~CBREAK);
                goto start;

        case T_WFLUSH:
                tp->t_tbuf.c_size -= tp->t_tbuf.c_count;
                tp->t_tbuf.c_count = 0;
        case T_RESUME:
                tp->t_state &= ~TTSTOP;
                goto start;

        case T_OUTPUT:
        start:
                if (tp->t_state&(TIMEOUT|TTSTOP|BUSY))
                        break;
                {
                        register struct ccblock *tbuf;

                        tbuf = &tp->t_tbuf;
                        if ( tbuf->c_ptr == NULL ||
                                tbuf->c_count == 0 ) {
                                if ( tbuf->c_ptr )
                                        tbuf->c_ptr  -= tbuf->c_size
                                                - tbuf->c_count;
                                if ( ! (CPRES &
                             (*linesw[tp->t_line].l_output)(tp)))
                                                break;
                        }
                        tp->t_state |= BUSY;
                        outb(addr + RTHR, *tbuf->c_ptr++);
                        tbuf->c_count--;
                }
                break;

        case T_SUSPEND:
                tp->t_state |= TTSTOP;
                break;

407          case T_BLOCK:
408                  tp->t_state &= ~TTXON;
409                  tp->t_state |= TBLOCK;
410                  if (tp->t_state&BUSY)
411                          tp->t_state |= TTXOFF;
412                  else
413                          outb(addr + RTDATA, CSTOP);
414                  break;
415
416          case T_RFLUSH:
417                  if (!(tp->t_state&TBLOCK))
418                          break;
419          case T_UNBLOCK:
420                  tp->t_state &= ~(TTXOFF|TBLOCK);
421                  if (tp->t_state&BUSY)
422                          tp->t_state |= TTXON;
423                  else
424                          outb(addr + RTDATA, CSTART);
425                  break;
426
427          case T_BREAK:
428                  outb(addr + RCNTRL,
                                 inb( addr + RCNTRL ) | CBREAK );
429                  tp->t_state |= TIMEOUT;
430                  timeout(ttrstrt, tp, HZ/4);
431                  break;
432          }
433  }
```

The code is defined as follows:

Line no.	Definition
366:	The *cmd* argument determines the action taken.
368-375:	The time delay for outputting a break has finished. Reset the flag TIMEOUT, which indicates there is a delay in progress and stop sending a continuous space. Then, restart output by jumping to **start**. A WFLUSH command resets the character-buffer pointers and the count.
376-378:	Either a line on which output was stopped is restarting, or someone is waiting for the output queue to drain. Reset the flag TTSTOP, indicating that output on this line is stopped, and start the output again by jumping to **start** (line 381).
380-383:	Try to output another character. If some delay is in progress (TIMEOUT), or the line output has stopped (TTSTOP), or a character is in the process of being output (BUSY), just return.

384-399: This code manipulates the character queue in order to output either a block of characters (by calling the **l_output** routine) or perform a single-character-output operation (in this example, the **outb**(K) routine).

Note that if the device is capable of outputting more than one character in a single operation, then this should be done, and the buffer pointer (**c_ptr**) and the count (**c_count**) should be adjusted appropriately.

403-405: To stop the output on this line, since there is no way to stop the character we have already passed to the controller, just flag the line stopped, and drop through.

407-414: To tell the device on the other end to stop sending characters, reset the flag asking to stop the line, and mark the line stopped. If the line is already busy, set the flag; otherwise, output a CSTOP character.

416-418: A process is waiting to flush the input queue. If the device hasn't been blocked, just return. Otherwise, drop through and unblock the device.

419-425: To tell the device on the other end to resume sending characters, adjust the flags. If the controller is sending a character, set the flag to send a CSTART later; otherwise, send the CSTART now.

427-431: To send a break, set the transmitter to continuous space, mark the line as waiting for a delay, and schedule output to be restarted later.

Chapter 5

Video Adapter Device Drivers

Writing a Video Adapter Driver 5-1
- Text and Non-Text Modes 5-3
- What the Video Driver Provides 5-3
- Hardware and Software Cursors 5-4
- How a Character Is Inserted 5-4
- Multiscreen Switching 5-5
- Memory Management in a Video Driver 5-6
- How a Video Driver is Installed 5-6
- Video Driver Notes 5-7

Video Driver Structures 5-8
- Multiscreen Structure 5-8
- Adapter Structure 5-11
- Keyboard Group Structure 5-13

Adapter Driver Routines 5-15
- Routine Descriptions 5-15

Writing a Video Adapter Driver

The device driver for the console, *vid.c*, contains a series of calls to eleven device-specific routines that a driver written for a video adapter must include. These routines perform initialization, character display, and special functionality. This chapter explains the functionality for each routine.

NOTE: This chapter should be used in conjunction with the sample video driver provided in the System V distribution software for the development system. The files are all in the */usr/lib/samples/pack.d/exvd* directory and are described as follows:

File	Description
exvd.c	Video adapter driver for a CGA board
m6845.c	Video controller-specific routines for *exvd.c*
m6845.h	Header file for the controller-specific routines
vidloops.c	Additional routines used by *exvd.c*

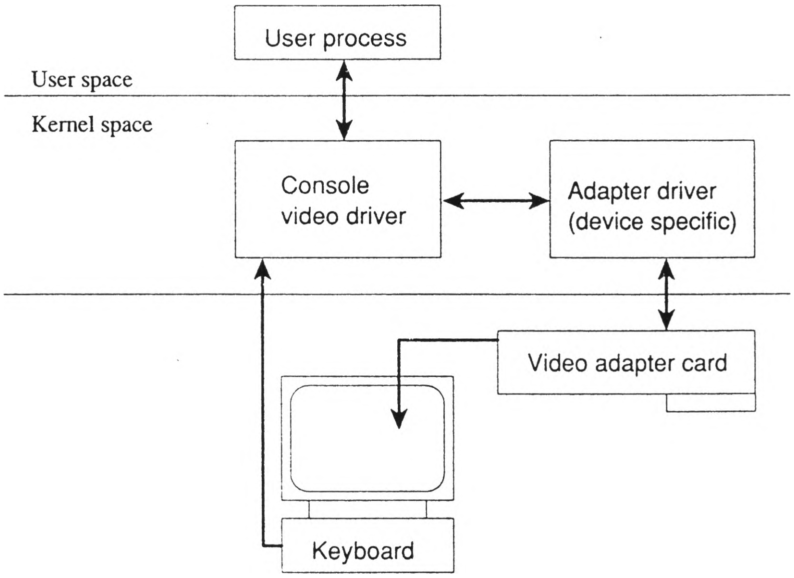
In addition, you should also examine the console header file, *sys/vid.h* for more information on the structures and variables discussed in this chapter.

In this chapter, the video console driver is called the *video driver*. The driver that you are writing or maintaining for the video adapter is called an *adapter driver*.

The video driver contains generic, device-independent routines. Separate adapter drivers provide device-specific functionality for the monochrome, CGA, EGA, and VGA adapters. If you have a video adapter card that uses another display medium or is non-standard, your driver only need include the device-specific routines. All other functionality for writing a video driver is handled for you by the video driver.

Writing a Video Adapter Driver

The following example shows the relationship between the user, the kernel, the video driver, the adapter driver, and the hardware:



Adapter Driver Routines Overview

Eleven predefined entry points are provided for adapter drivers. These routines are as follows:

- *Initialization Routines* — **xxcmos**, **xxinit**, and **xxinitscreen**
- *Data Handling Routines* — **xxscroll**, **xxclear**, **xxcopy**, and **xxpchar**
- *Special Routines* — **xxscurs**, **xxioctl**, **xxsgr**, and **xxadapctl**

As with all device drivers, you replace the **xx** prefix with the unique two-, three-, or four-letter identifier for your particular driver. As previously stated, there are supplied adapter drivers for standard monochrome (mono), CGA, EGA and VGA adapters.

Text and Non-Text Modes

When describing how the video system works, two modes are used: *text* and *non-text*. Text mode is used for displaying normal text and allows switching between multiscreens. Non-text mode is used for bit-mapped graphics and do not allow multiscreen switches.

Routines Required for Each Mode

The following table shows which adapter routines are required for each mode:

Routine	Text Mode	Non-Text Mode
xxadapctl	required	required
xxclear	required	not used
xxcmos	required	required
xxcopy	required	not required
xxinit	required	required
xxinitscreen	required	required
xxioctl	required	required
xxpchar	required	not required
xxscroll	required	not required
xxscurs	required	not required
xxsgr	required	not required

What the Video Driver Provides

The video driver contains the normal character driver routine interface: **xxread, xxwrite, xxioctl, xxopen**, and **xxclose**. Requests to the **xxwrite** routine are processed by an ANSI parser. The parser calls the adapter driver routines to change the text display, move the cursor, change text color, and so on.

The console video driver uses the multiscreen number to index into an array of *struct mscrn* to find the address of the current multiscreen structure - *msp*. Assuming that the screen is in text mode it proceeds, otherwise it drains and discards the output without generating an error.

Hardware and Software Cursors

Two cursors are discussed in this chapter, a *software cursor*, and a *hardware cursor*. The hardware cursor is the blinking underbar or block that is displayed on the screen. The software cursor is designated by the **mv_row** and **mv_col** fields of the *mscrn* structure.

When text is displayed by a call to the adapter driver, the software cursor becomes the output point for displaying text. When the video driver calls the **xxpchar** routine of the adapter driver, the buffer of characters passed to **xxpchar** is displayed in successive positions on the screen, The software cursor is updated only by the video driver, but not by the adapter driver.

When the video driver completes an **xxwrite** request and at other special times like after cursor motion or an escape sequence, the **xxscurs** routine of the adapter driver is called to update the hardware cursor with the software cursor position.

To give the hardware cursor the same value as the software cursor, **mv_row** and **mv_col** are set, and **xxscurs** is called to update the display.

How a Character Is Inserted

To illustrate how an adapter driver works in conjunction with the video driver, this section describes how a character is inserted using an adapter driver. The adapter driver is called by the video driver. In the described example, a character is inserted at the current position into the fifth multiscreen, on a computer that has two video adapters. The sequence is as follows:

1. The console video driver uses the multiscreen minor device number to index into an array of *struct mscrn* to find the address of the current multiscreen structure — *msp*. Assuming that the screen is in text mode it proceeds, otherwise it drains and discards the output without generating an error.

2. The software cursor position is contained in the *mv_row* and *mv_col* fields of the *mscrn* structure.

3. The value of *msp* is stored in *mv_adapter->v_curscrn* so that the adapter routines can determine if this multiscreen is being displayed or not.

4. **xxcopy** is called to move the characters beginning at *mv_row, mv_col* one position to the right.

5. The character is drawn on the screen by a call to **xxpchar** and the cursor is moved by a call to **xxscurs**.

Multiscreen Switching

This dialogue of the video driver and the adapter driver is similar for each function handled by the video driver. In a similar fashion, a multiscreen switch occurs as follows:

1. The video driver uses the multiscreen number to index into an array of *struct mscrn* to find the address of the current multiscreen structure — *msp*. The video driver then tests to see if the screen is in *text* mode, and if it is, proceeds as explained in the steps that follow; otherwise the video driver announces an error by causing the speaker to beep.

2. The video driver allocates a screen save area. Screen save areas are allocated at screen initialization time, are always available for use by adapter drivers, and are reallocated on mode switches because the size of the screen may change. (Mode switches cause screen data to be cleared.)

3. The current screen display is copied to the save area when the video driver calls the adapter driver with this statement:

   ```
   (*(mv_adapter->v_adapctl))(msp, AC_SAVSCRN, arg)
   ```

   The *arg* argument points to the save memory area.

4. The saved image of the video screen for the newly selected multiscreen is copied into display RAM when the video driver calls the adapter driver with this statement:

   ```
   (*(mv_adapter->v_adapctl))(msp, AC_RESSCRN, arg)
   ```

   The *arg* argument points to the save memory area.

Memory Management in a Video Driver

The **vas** routines described on the **vas**(K) manual pages are used to share memory with a user process, for example, so that the user process has the video frame buffer in its user address space.

The **sptalloc**(K) routine is used to map video adapters into memory space. For example, to map a card into memory, the following **sptalloc** call is used:

```
sptalloc(1, PG_P, 0xB8000, 1);
```

This call maps 1 memory page at address 0xB8000 and it indicates that the page should always be present, not swappable (PG_P). The fourth argument, 1, has no effect. If memory is not available, **sptalloc** sleeps until it is available.

How a Video Driver is Installed

A video driver is installed using the following procedure:

1. Compile your program to create an object file using the information described in Chapter 6.

2. Copy the driver object file to the */etc/conf/pack.d* directory.

3. Edit */etc/conf/cf.d/mvdevice*. Refer to **mvdevice**(F) for information.

4. Edit */etc/conf/cf.d/mdevice*. Refer to **mdevice**(F) for information.

5. Edit */etc/conf/sdevice.d/*<driver-name>. Refer to **sdevice**(F) for information.

6. Run *link_unix* to build a new kernel. Refer to Chapter 6 for more information.

Video Driver Notes

1. The adapter driver can and does write directly into the *mscrn* structure, as it can for any other structures held by the video driver. These structures are described in *sys/vid.h*, and the principle structures for multiscreens, adapter drivers, and keyboard groups are described in the section "Video Driver Structures" in this chapter.

2. The **xxadapctl, xxclear, xxcopy, xxpchar, xxscroll, xxscurs,** and **xxsgr** routines can be called at interrupt time (due to echo and screen switches being done at interrupt time. The **xxcmos, xxinit,** and **xxinitscreen** routines are called from an **xxinit** routine, and the **xxioctl** routine is called from user context only.

3. The video driver allocates the amount of screen memory that was allocated during a previous call to **xxadapctl** with the AC_SAVSZQRY command. This memory is available at the multiscreen's save-screen pointer and is always available.

Video Driver Structures

The video driver uses main structures in which an adapter driver can read or write. These structures are:

- Multiscreen structure (*mscrn*) — Contains state information for each multiscreen. One instance of the structure is required for each multiscreen.

- Adapter structure (*adapter*) — Contains pointers for each adapter driver routine. One instance of the structure is required for each adapter driver.

- Keyboard group structure (*kbgrp*) — Contains pointers for each keyboard directly attached to the computer (not to serial lines). One instance of the structure is required for each keyboard.

Multiscreen Structure

The interaction in the adapter driver depends heavily on access of the multiscreen data structure, *mscrn*. The members of this structure are as follows:

Type and Field	Description
`struct kbgrp *m_grp;`	`/* Multiscreen group header */`
`int m_num;`	`/* Screen number */`
`struct tty *m_tty;`	`/* tty structure */`
`struct adapter *mv_adapter;`	`/* Ptr to the screen's adapter */`
`vseg_t mv_savscrn;`	`/* Save screen RAM */`
`unsigned mv_savsz;`	`/* Size of save screen used */`
`faddr_t mv_smap;`	`/* Font value translation table */`
`uchar_t mv_font;`	`/* ANSI font, one of 0, 1, or 2 */`
`ushort mv_tmpool;`	`/* Used by ANSI */`
`ushort mv_row, mv_col;`	`/* Cursor position */`
`ushort mv_rsz, mv_csz;`	`/* Text screen size */`
`ushort mv_csstate;`	`/* Ctl sequence state */`
`ushort mv_csparam[NCSPARAMS];`	`/* Ctl sequence parameters */`
`uchar_t mv_csindex;`	`/* Ctl sequence parameter index */`
`uchar_t mv_cstyp;`	`/* Cursor type */`

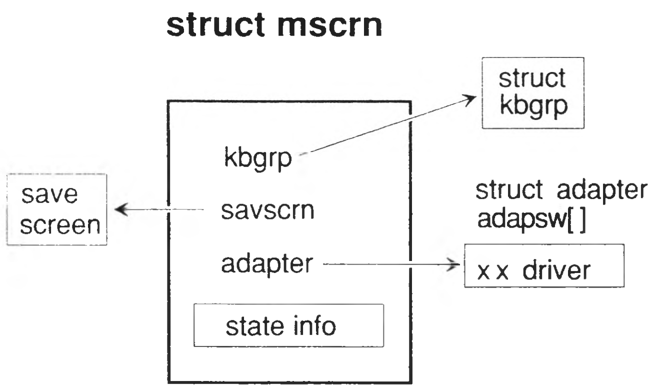
Type and Field	Description
struct colors mv_norm,	/* Normal attributes */
mv_rev,	/* Reverse video attributes */
mv_grfc;	/* Graphic character attributes */
uchar_t mv_ovscan;	/* Border color */
char mk_qstr[MAXFK];	/* Work space for func. key conf */
strixp_t mk_strix;	/* Function key string table pointer */
strtabp_t mk_strtab;	/* Function key string text table */
uchar_t mk_keylock;	/* Caps/num/scroll lock */
ushort mb_time;	/* Bell duration 1/10 secs */
ushort mb_freq;	/* Bell frequency (pitch) */
ushort mf_savrow;	/* ^[[7 save current row & col */
ushort mf_savcol;	/* ^[[8 restore row & col */
ushort mf_snd_row;	/* Row for sendscreen */
ushort mf_snd_col;	/* Column for sendscreen */
ushort mf_status;	/* Terminal status bit field */
ushort mf_xtraopenf;	/* Flags for systty support */
uchar_t adp_area[32];	/* See note that follows */

NOTE: *adp_area* provides hardware fields for each multiscreen for use by adapter drivers. This area can be used in any manner.

struct mscrn is a structure that is replicated for each individual multiscreen. Each structure contains state information for the multiscreen (current color, mode, and so on) and a series of pointers.

mscrn contains a pointer to *struct kbgrp*, the keyboard group that the multiscreen belongs to. Note that all multiscreens on the main system monitor have the same keyboard group. *mscrn* also contains *mv_savscrn* which points to the RAM memory location where the screen image is saved when the multiscreen is not active. Also of interest in *mscrn* is a pointer to *struct adapter* which contains information about the adapter driver.

The relationship between *struct mscrn* and the other parts of the driver is shown in the following diagram:

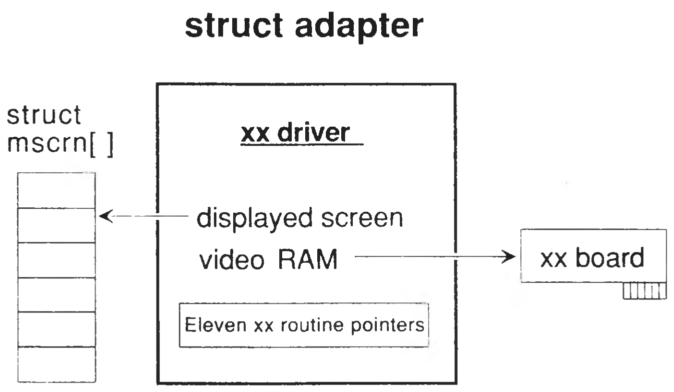


Adapter Structure

struct adapter is a structure that is replicated for each adapter driver and contains pointers to each of the eleven routines that an adapter driver should have, and other pointers to the *struct mscrn* of the current multiscreen and to the video RAM of the card associated with the adapter driver.

Type and Field	Description
`char *v_name;`	`/* Name of the adapter ie "CGA" */`
`ushort v_type;`	`/* Basic type such as MONO, CGA,... */`
`ushort v_oem;`	`/* Non-standard vendor name */`
`struct mscrn *v_curscrn;`	`/* Pointer to the current screen */`
`ushort v_initrc;`	`/* Saved return value of v_init */`
`int (*v_init)();`	`/* Init the adapter driver */`
`int (*v_cmos)();`	`/* Extra cmos type checking */`
`int (*v_initscreen)();`	`/* Init a multiscreen on the adapter */`
`void (*v_scroll)();`	`/* Scroll screen up or down */`
`void (*v_copy)();`	`/* Copy data between screen areas */`
`void (*v_clear)();`	`/* Clear any portion of the screen */`
`void (*v_pchar)();`	`/* Put a string of characters */`
`void (*v_scurs)();`	`/* Set the active and cursor position */`
`void (*v_sgr)();`	`/* Set graphic rendition; see ANSI X.64 */`
`int (*v_ioctl)();`	`/* Adapter specific ioctl handler */`
`int (*v_adapctl)();`	`/* Adapter specific misc other stuff */`
`vseg_t v_videoram;`	`/* Kernel pointer to video ram */`
`paddr_t v_paddr;`	`/* Physical address of video ram */`
`ulong_t v_size;`	`/* Size of of video ram in bytes */`

The following is a graphical representation of the relationship between *struct adapter* and the other hardware and software that comprise the video system:

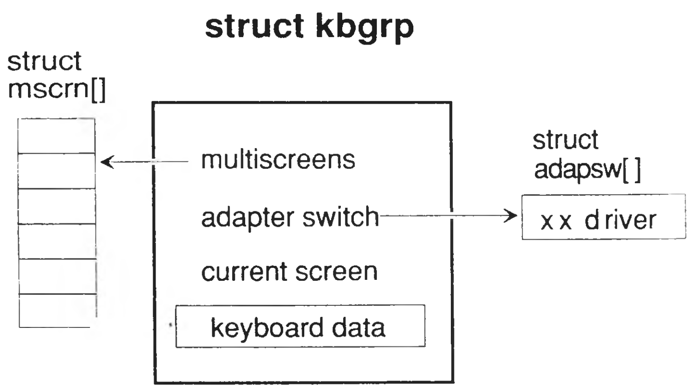


Keyboard Group Structure

struct kbgrp defines a keyboard group. This structure is replicated for every keyboard attached directly to your computer. (Keyboards on terminals attached via serial lines are not included.) The *struct kbgrp* has a pointer to each multiscreen that accepts input from the keyboard, a pointer to *struct adapter*, a variable indicating the current multiscreen, and any pertinent keyboard data.

Type and Field	Description
`struct mscrn *kg_m0;`	`/* multiscreens[] for the keyboard grp */`
`struct adapter *kg_a0;`	`/* adaptsw[] for the keyboard grp */`
`int (*kg_in)();`	`/* keybd grp line displn in() routine */`
`struct map *kg_memmap;`	`/* keybd grp multiscreen save area map */`
`short kg_dtoa[MAXADAPTERS];`	`/* device # to adapter index table */`
`short kg_curscrn;`	`/* current screen */`
`short kg_maxscrn;`	`/* maximum valid screen in keyboard grp */`
`struct keymap *kg_keymap;`	`/* scan code translation map */`
`char kg_rabuf[MAXRA];`	`/* keyboard scan code read ahead buffer */`
`uchar_t *kg_rafp;`	`/* read ahead buf front pointer */`
`uchar_t *kg_rabp;`	`/* read ahead buf back pointer */`
`uchar_t kg_break;`	`/* was last scan code a break? */`
`short kg_down;`	`/* which state keys are down? */`
`uchar_t kg_state;`	`/* local keyboard state */`
`uchar_t kg_gblklk;`	`/* global key lock keyboard state */`
`uchar_t kg_kbmode;`	`/* keyboard mode AT vs. XT */`
`int (*kg_scrsend)();`	`/* kb command sender (leds) */`
`int (*kg_scrdrain)();`	`/* clear any keyboard data */`
`int (*kg_scrgetkey)();`	`/* fetch key from keyboard */`
`int (*kg_scrmode)();`	`/* put kb in AT or XT mode */`
`int (*kg_bell)();`	`/* activate bell */`
`int kg_extmode;`	`/* extended mode flag */`
`int kg_altseq;`	`/* ALT key sequence */`

The following picture is graphic representation of the relationship between *struct kbgrp* and the other software for the video driver:



Adapter Driver Routines

The following is a description of the eleven driver routines that you provide when writing an adapter driver. The following table indicates in which context each routine must be equipped to operate:

Interrupt or User Context	*Only User Context*	*Initialization Only*
xxadapctl	**xxioctl**	**xxcmos**
xxclear		**xxinit**
xxcopy		**xxinitscreen**
xxpchar		
xxscurs		
xxsgr		

Routine Descriptions

Syntax: **xxcmos(prip, secp)**
int *prip;
int *secp;

Purpose: This routine reads the hardware characteristics and specify whether the video card in question is present and should be designated as the primary card. This routine is required except for monochrome and CGA adapter drivers. The adapter driver (your code) sets the driver type in the word pointed to by **prip*.

In addition, you can use this routine to do any other checks or start-up operations you desire. For example, the EGA driver uses **egacmos** to determine if the EGA card is present and if it is, to read the switches on the card. If the switches indicate that the card is present and that the card should be set to be the primary card, then "EGA" is returned in the word pointed to by ***prip**.

If a serial console is not attached to the system, the system boot message appears on the primary adapter and all multiscreens are attached to the primary adapter.

Parameters: The pointers *prip* and *secp* point to the primary and secondary video devices to be used. Your driver can designate itself as the primary device by assigning the adapter type (**v_type**) from the *adapter* structure into *prip*. Likewise, you can also assign your driver to be secondary by assigning its value to *secp*.

Syntax: **xxinit(adp)**
struct adapter *adp;

Purpose: This routine is called only once; at driver initialization time. Use this routine to do any initialization to your hardware that needs to be done.

Parameters: *adp* points to the adapter that is to be initialized.

Return: This routine can return two flags: AI_PRESENT and AI_COLOR, masked together in a bitwise OR. AI_PRESENT indicates that device is present, and AI_COLOR indicates that color is to be supported.

Syntax: **xxinitscreen(msp)**
struct mscrn *msp;

Purpose: This routine is called when moving a multiscreen to a new adapter, at system initialization time, and at the last close of a multiscreen when in graphics mode.

Parameters: *msp* points to the multiscreen that is being initialized.

Syntax: **xxscroll(msp, cnt)**
struct mscrn *msp;
int cnt;

Purpose: This routine scrolls the screen up or down. Upward scrolling is obtained by specifying *cnt* as a positive integer. Downward scrolling is obtained by specifying *cnt* as a negative integer. In either case, the integer value of *cnt* is the number of lines to be scrolled.

Parameters: *msp* points to the multiscreen that is being scrolled. *cnt* is the number of lines to be scrolled.

Syntax: **xxclear(msp, drow, dcol, cnt)**
struct mscrn *msp;
int drow, dcol, cnt;

Purpose: This routine clears any portion of the screen from one character to the entire screen. The symbol positions are cleared with the space (0x20) font character using the current attributes (such as reverse video.)

Parameters: *msp* points to the multiscreen that is being cleared. *drow* and *dcol* are the destination row and column. *cnt* specifies the number of displayed symbols to be cleared.

Syntax: **xxcopy(msp, drow, dcol, srow, scol, cnt)**
struct mscrn *msp;
int drow, dcol, srow, scol, cnt;

Purpose: This routine copies data from one portion of the screen to another. For example, if a word on the screen is deleted, perhaps by an editor command, this routine implements the escape sequence the editor would use to move the remaining text over, filling in the blank space. Attributes (if any) are also copied.

Parameters: *msp* points to the multiscreen that is being copied. *drow* is the row where receiving is to begin. *dcol* is the column (space) where receiving is to begin. *srow* is the source row from where the copied data is to be drawn. *scol* is the source column (space) in *srow* from where the information is to be drawn. *cnt* is the number of characters to be copied.

Syntax:

xxpchar(msp, bp, cnt)
struct mscrn *msp;
char *bp;
int cnt;

Purpose: This routine writes data beginning at the current software cursor position. This routine does not move the hardware cursor; the **xxscurs** routine is used for this purpose.

Parameters: *msp* points to the multiscreen that is being written to. *bp* is the buffer pointer supplying the data. *cnt* is the number of characters to be written.

Syntax:

xxscurs(msp)
struct mscrn *msp;

Purpose: This routine updates the hardware cursor to be the same value as that of the software cursor. The position of the software cursor is contained in the **mv_col** and **mv_row** fields of the *mscrn* structure.

Parameters: *msp* points to an instance of *mscrn* that contains the position of the software cursor.

Syntax: **xxsgr(msp, sgr)**
struct mscrn *msp;
int sgr;

Purpose: This routine provides support for the ANSI standard video functionality. *sgr* stands for set graphics rendition. In this case, a ``rendition'' means a video effect, such as reverse video, underlining, or blinking. You should write this routine to accept ANSI standard video instructions and convert them to whatever codes your hardware requires to perform the standard video functions.

Parameters: *msp* points to the multiscreen receiving the instructions. *sgr* is the ANSI code for the desired graphics rendition.

Standard values of *sgr* that are passed to your **xxsgr** routine are: (missing values are not supported)

SGR_NORMAL	0	/* return attributes to normal */
SGR_BOLD	1	/* also called INTENSE */
SGR_PRCOLORS	2	/* PR's set the normal colors
SGR_PRBLKCTL	3	/* PR's blink bit control */
SGR_UNDERL	4	/* underline */
SGR_BLINK	5	/* blink */
SGR_REVERSE	7	/* reverse video */
SGR_CONCEALED	8	/* hide characters */
SGR_FONT	10	/* fonts 0 through 9 w/ 10-19 */
SGR_RES1	26	/* ANSI reserved */
SGR_FORECOLOR	30	/* ANSI foreground */
		/* colors: 30 to 37 */
SGR_RES2	38	/* ANSI reserved */
SGR_BACKCOLOR	40	/* ANSI background */
		/* colors: 40 to 47 */
SGR_RES3	48	/* ANSI reserved */

Foreground and background colors are defined in the ANSI color table information on the **screen**(HW) manual page.

Syntax:	**xxioctl(msp, cmd, arg, mode)**
	struct mscrn *msp;
	int cmd;
	char *arg;
	int mode;

Purpose:	This routine provides support for any I/O control commands (ioctls) that you may want to support. In practice, you can use this routine to support standard ioctls for your hardware or you can create your own ioctls.
Parameters:	*msp* points to the multiscreen receiving the instructions. *cmd* is the ioctl command. The standard ioctls are CONS_GET and MAPCONS. These stand for CONSOLE GET and MAP CONSOLE. *arg* is any arguments to the ioctl command. *mode* is the new mode (such as graphics or text) for your card. See the **screen**(HW) manual page for any additional ioctls you may need to support.

Syntax: **xxadapctl(msp, cmd, arg)**
struct mscrn *msp;
int cmd, arg;

Purpose: This routine provides support for adapter specific functionality between the video driver and your adapter driver.

Parameters: *msp* points to the multiscreen receiving the instructions. *cmd* is the ioctl command. *arg* is any arguments to the ioctl command. Standard values of *cmd* passed to your routine are:

AC_BLINKB	0	/* clear or set the blink bit */
AC_FONTCHAR	1	/* display font character */
AC_DEFCSR	2	/* define cursor type */
AC_BOLDBKG	3	/* turn on intense bg color */
AC_DEFNF	10	/* define normal foreground */
AC_DEFNB	11	/* define normal background */
AC_ONN	12	/* begin using normal colors */
AC_DEFRF	13	/* define reverse foreground */
AC_DEFRB	14	/* define reverse background */
AC_ONR	15	/* turn on reverse colors */
AC_DEFGF	16	/* define graphic foreground */
AC_DEFGB	17	/* define graphic background */
AC_ONG	18	/* turn on graphic colors */
AC_SETOS	30	/* set overscan colors */
AC_PRIMODE	100	/* return primary text mode */
AC_SAVSZQRY	101	/* return size (bytes) of state */
AC_SAVSCRN	102	/* save screen */
AC_RESSCRN	103	/* restore screen */
AC_CSRCTL	104	/* arg=0 hide cursor, arg=1 show cursor */
AC_USERFONT	105	/* load or dump the soft font */
AC_IOPRIVL	106	/* grant or revoke I/O privilege */
AC_SOFTRESET	107	/* reset text mode (keep colors)*/
AC_SENDSCRN	108	/* write screen chars to stdin */

The arguments to the above commands will be such values as are appropriate. For example, an argument to AC_FONTCHAR would be the new font to be used.

Chapter 6

Compiling and Linking Drivers

Compiling, Configuring, and Linking Drivers 6-1
Selecting a Prefix 6-1
Compiling a Device Driver 6-2
System Configuration 6-2
Linking The Kernel 6-5

Driver Debugging 6-6
Booting the New Kernel 6-6
General Debugging Hints 6-7
Vector Collision Considerations 6-9
Note on ps 6-10

Notes On Preparing a Driver for Binary Distribution 6-11
Naming Guidelines 6-11
Style Issues for User Prompting 6-11
Shielding Against Configuration Changes 6-12
Preparing Drivers to Use custom(C) 6-13
Driver Installation Script Overview 6-15

Compiling, Configuring, and Linking Drivers

To make your driver source code part of the kernel, a series of steps should be followed. This chapter describes each step in greater detail. The steps are:

Step	*Purpose*	*Completed?*
1	Select a unique prefix	
2	Compile the driver source	
3	Ensure that the device is installed	
4	Adjust kernel tunable parameters	
5	Check for address or interrupt vector conflicts	
6	Get the next available major number	
7	Alter configuration files	
8	Relink the kernel	
9	Create a device node	
10	Invoke the new kernel	

Selecting a Prefix

The prefix is a two-, three-, or four-character unique name used to identify a driver. The prefix name is used to preface all routine names in a driver. The **configure**(ADM) command does not permit a driver to be installed if its prefix is already in use. Display the */etc/conf/cf.d/mdevice* file to see the existing prefix names. You should select a prefix name that represents the device your driver is associated with. If you are installing a driver that has a conflicting prefix, you can adjust the prefix with **configure -h**. (**configure** is in */etc/conf/cf.d*; change directory to run **configure**.)

Compiling a Device Driver

Use the C compiler to compile C source code, or the assembler to create an object module from assembler source. Use the **cc**(CP) or **masm**(CP) commands.

The **cc** command line must contain the following switches:

-c	Create a linkable object file.
-K	Disable stack probes.
-Zp4	Align program on quad-word boundaries.
-DINKERNEL	Required for conditional code in standard header files

It should also contain -M3 for the 80386 processor.

For device driver subroutines written in macro-assembler language, the **masm** command line should contain the following switch:

-Mx	Preserves lower case in output. Required for the linker to be able to resolve external declarations to C functions.

An appropriate **cc** or **masm** command line produces a corresponding ''.o'' module. For example, *scsi.c* becomes the object module *scsi.o*.

System Configuration

System configuration is the process of placing references to your driver's main functions in various tables. Since the existing parts of the kernel do not know what the functions in your new driver are called, driver functions are referenced by indirect calls into the configuration tables.

Composing the driver's configuration command is discussed in **configure**(ADM). **configure** insulates you from potential changes to the configuration files, and allows you to use the same procedure to configure your driver as the end-user who receives your driver in binary form. (**configure** is in */etc/conf/cf.d*; change directory to run **configure**.)

Determining the Vector Number

You must determine your interrupt vector number so you can inform the kernel that your driver should be called when an interrupt is pending on that vector. This information is highly machine and configuration dependent.

For the 80386 mapped kernel, your peripheral device can interrupt on one of the request lines of either a master interrupt controller or single slave interrupt controller. The slave controller is connected to master request line 2. Your vector number does not correspond directly to the bus request numbers. Instead, it is mapped to logical vector numbers which allow for the presence of slave interrupt controllers connected to the main one.

The index of the appropriate vector is determined as follows:

1. If the vector used is on the master controller, just use the vector number directly. These numbers range from 0 to 7.

2. If it is on a slave controller (only one is currently supported for the 80386, on master request line 2), take the request line used on the slave controller and add decimal 7. The result is used in */etc/conf/cf.d/sdevice* file.

For example, if your device uses request line 3 on the slave controller, you would specify decimal 10 in the *sdevice* file. The slave controller vector numbers are:

Slave Controller Request Line	Specify this Vector Number In the *sdevice* File
1	8
2	9
3	10
4	11
5	12
6	13
7	14
8	15

Vector Manipulation for Preconfigured Device Drivers

In preconfigured drivers, entry points have been provided for all necessary routines (**xxopen, xxclose**, and so on) except for the interrupt handler. The interrupt handler can be configured with the **configure**(ADM) command if a vector conflict occurs; otherwise, the interrupt vector that you establish when you create your driver should suffice. A vector conflict can be detected with the **configure -V** option.

See the section entitled ''Vector Collision Considerations'' for more information on the selection of interrupt vectors.

Using configure

Before **configure** can be run, you need to know an unused major device number for your device, the vector or vectors on which your device interrupts, and the list of routines in your driver that must be added to the configuration tables.

The **configure** utility enforces the rules that all routines in the driver begin with a common prefix and that the prefix be two-, three-, or four-letters long. If your driver prefix is incorrect or inconsistent, change it.

Please read **configure**(ADM) and ''Adding Device Drivers with the Link Kit'' in the *System Administrator's Guide*. This chapter contains a detailed description of how to create a **configure** command line for a driver binary. Authors of drivers have an advantage in that they do not have to discover the names of the routines; the names that must be presented to **configure** are those chosen for the routines that have so far been described, such as the name of a character driver's **write** routine. (**configure** is in */etc/conf/cf.d*; change directory to run **configure**.) Maintain a backup copy of the */etc/conf/cf.d/mdevice* file, and the */etc/conf/sdevice.d* and */etc/conf/pack.d* directories. If a mistake occurs restore the old files or reinstall the Link Kit.

Also note that **configure** requires that block drivers have a *xxtab* structure, and indeed, the vast majority of block drivers do. If you are writing a block driver without an interrupt routine, simply declare a *struct iobuf xxtab* within your driver.

Linking The Kernel

Your Link Kit contains the */etc/conf/cf.d/link_unix* shell script for linking the kernel.

Before linking your driver, ensure that your driver's object file is in the */etc/conf/pack.d* directory, and then change directory to */etc/conf/cf.d*.

Link your driver into the kernel by entering:

```
./link_unix
```

Driver Debugging

The following sections contain information on getting a driver to run, and what to look for if it doesn't.

Booting the New Kernel

Halt your system by entering:

/etc/shutdown

You see the ''** Normal System Shutdown **'' message. Press return to see the boot prompt:

```
Boot
:
```

If you press RETURN, or simply do nothing, the new operating system image is loaded and started. Bring up the new kernel at the **Boot:** prompt by entering **unix**:

```
Boot
: unix
```

The system boots from the ''new'' kernel. If you need to recover the old kernel, enter **unix.old** at the **Boot:** prompt:

```
Boot
: unix.old
```

General Debugging Hints

Debugging a device driver is more an art than a science. This section touches on some of the more useful techniques to try if your driver isn't working.

1. Make sure that you are actually talking to your driver.

   If you get errors such as "no such device" (ENODEV) when you try to access your driver, the problem can be the device node itself.

   Check your major device number correspondence.

2. Make sure your device registers are where you think they are.

   The effect of accessing a nonexistent port address varies from machine to machine, but, for example, on the IBM XT or AT you can read values using **inb**(K) from nonexistent hardware and receive no error code, just a random value.

   Since at least some of the I/O ports on most peripheral controllers are both read and write, you should make sure that you can write to one of your device's registers using **outb**(K), then read back the value you've written using **inb**(K). Even when none of the registers are read/write, as is true on some mouse controllers, you can at least read from one of the status registers using **inb**(K), and make sure that the result is reasonable.

3. Work towards getting simple I/O from the driver first, complex I/O later.

   Character devices are usually easier to write to than to read from. For a serial or a printer driver, your first test will probably be to echo "hello, world" to the device, or something equally simple and traditional.

   Block devices are generally easier to read from than to write to, since you have to read back the block you've written to know if you've written it successfully. Many block devices have a "get drive parameters" command, or something similar, which is even more basic than either reading or writing.

4. Use kernel **printf**(K) or **cmn_err**(K) for debugging, *except* when in an interrupt routine. In an interrupt routine, put your debug messages in a static structure and then display them to the console in task time.

Although you shouldn't overuse **printf** (in a finished driver, **printf** should only be used for unrecoverable errors), it can be an invaluable debugging tool.

Coupling **printf** with #ifdef DEBUG statements and a global "debug level" flag, you can tailor the verbosity of your debug output to the situation at hand.

For example, you may have two debug levels:

```
#ifdef DEBUG
  if ( mydebugflg > 0 )
   printf("got to myopen()\n");
  if ( mydebugflg > 2 )
    printf("open parameters: dev=%x", dev);
    printf("flag=%x bc=%x\n",flag, bc);
#endif
```

There are occasional situations where a **printf** can change peripheral timing enough to change the behavior in question, but these cases are fairly rare.

5. Use **getchar**(K) to stop kernel output and to set debug levels.

   Kernel **getchar** is similar, though not quite the same, as the standard I/O library routine of the same name. Kernel **getchar** returns a single character from the keyboard. The character is automatically echoed. The only other processing done on this character is to map RETURN to RETURN/LINE FEED on output. When you have many lines of kernel **printf**(K) output, inserting **getchar** statements into your driver is one of the better ways to regulate the **printf** output flow.

   A second use of **getchar** is to set the level of debugging. For example, in the example above, you could place two lines of code such as:

   ```
   mydebugflg = getchar();
   mydebugflg -= '0';
   ```

   shortly after the beginning of the open routine, to set the current value of *mydebugflg* to anywhere between 0 and 9.

   Note that **getchar** may not work at interrupt time for interrupt routines of certain priorities.

6. Poll before you use interrupts. Polled drivers are best first approximations for block devices such as disks. For serial drivers, a polled interface may help you decide how to write to the device.

However, be forewarned that performing polled reads will make the system unusably slow.

Often the hardest driver routine to get right is the interrupt routine. You can expedite this process by first writing a polled driver: one that busy-waits until the request you made has completed, and then returns status. However, do not leave any busy-wait loops in the finished driver!

7. Use **spl7**(K) as a debugging aid. Ensure that the use of **spl7** is for debug purposes only. Remember that any code protected by **spl7** is also blocking the clock ticks from occuring. Always use any **spl** routine in conjunction with **splx**. All **spl** routines are described on the **spl**(K) manual page in this guide.

   Sometimes, a driver can be difficult to debug because higher priority interrupts get in the way. A call to **spl7** shields driver code from any interruptions by the other devices on the system.

8. Be patient. Drivers are complex. So much so, that writing a 300-line device driver takes even an experienced driver-writer several times longer than a utility program of the same length. Don't worry if your driver takes a while to perfect.

Vector Collision Considerations

Design a driver with care when selecting a hardware interrupt vector, because conflicts can occur between drivers over interrupt vector usage.

80386-based systems use 2 8259 programmable interrupt controllers. The mapped kernel currently leaves only vectors 9-12 and 15 (bus leads IRQ9-12 and IRQ15) unused. These vectors are also safe to use for devices whose drivers are written using **spl5**.

If it is necessary to use one of the other vectors, there are two configuration alternatives:

1. Replace the device driver already using the vector.

2. Provide a special-purpose interrupt handler that ''knows'' that the vector is shared and takes appropriate precautions.

The first alternative is recommended, but is not always possible. There are problems with the second alternative, because there is no way to prevent the loss of interrupts which can occur when competing with an arbitrary device.

The problem is that the 8259 interrupt controller detects an interrupt request only when the request line changes state from off to on (called *edge-triggered* mode). If all sources for the interrupt request line are not off at the same time after entry to the interrupt service routine, no further *rising edge* on the request signal is detected, and so no more interrupts are seen on that vector until all the sources for the interrupt request line are turned off. The state of the interrupt request line cannot be determined directly from the interrupt controller chip, so the determination must be made by device-specific means for all devices sharing the vector.

However, sharing is possible for those devices that interrupt only following a request from the CPU. Disk drivers, tape drivers, and other such device drivers can ``time out'', using **timeout**(K), when waiting for a response to a request, and, upon time out, examine the device to determine if the operation is complete. This approach saves your driver from lost interrupts, but the device with which you share a vector is only immune if it is written using **timeout** as well.

This approach is far from practical for use with devices such as serial communication lines, which can cause interrupts at any time, out of the control of the system using the device. The granularity of control available with **timeout** is far too slow for all but the slowest of communication lines (approximately 110 to 200 baud).

This does not mean, however, that each serial line requires its own interrupt vector. Some serial boards provide enough pollable state information to allow the serial interrupt routine to loop until none of its controlled devices is posting an interrupt. In this example, the key is that a single interrupt routine controls all of the multiple devices on a single vector.

Note on ps

If you change to an alternate name for your kernel, such as *unix.new*, **ps**(C) does not work correctly unless you specify the **-n** flag and the pathname of the kernel you are using.

During debugging it is useful for the device driver writer to display the *address* argument of **sleep**(K). Use the shell command **ps -el** to identify which processes are sleeping in your driver by examining the values reported in the WCHAN column.

See **ps**(C) in the *User's Reference* manual for more information.

Notes On Preparing a Driver for Binary Distribution

Naming Guidelines

The two- to four-letter name that prefixes all of your driver's routines should describe what kind of a driver it is, as best as is possible in such limited space. For example, the current serial I/O driver uses routines beginning with ''sio'', and the parallel driver uses routines beginning with ''pa''.

Preconfigured drivers have had their names reserved in advance. If you are writing a driver for a device that a user might have more than one of, such as an add-on hard disk driver, you might want to be a bit more obscure to prevent later naming conflict. For example, the driver for a techno-babble hard disk might begin its routines with the prefix ''tbhd''.

Style Issues for User Prompting

Most currently configured devices print out a short message in their initialization routines using **printcfg**(K) to notify the user that they are installed. This message must be terse. All the extra drivers that a user could possibly want, combined, should not generate enough messages to scroll the boot-up copyright message off the screen.

For example, this is an appropriate message:

```
Device   Address         Vector   dma   Comment
------   -------         ------   ---   -------
serial   0x3F8-0x3FF      04      --    type=std ports=1
```

Note that the labels (Device, Address, Vector, and so on) are provided at boot time; you need only supply a line with information specific to your driver. Refer to the **printcfg**(K) manual page for more information on the format of an initialization message

Shielding Against Configuration Changes

Do not write a driver that relies on particular configuration parameters, for example a certain major device number or interrupt vector. Avoiding such ''hardcoded'' assumptions helps prevent collisions with other drivers, and insulates the driver from system configuration changes.

Drivers should not, and do not need to be aware of their own major device number. However, the device number contains the major number if you want to display it. Use **major**(K) to extract the major number from the device number.

Very few drivers have ever needed to know this information, but those that did fell into two categories: drivers performing some form of physical I/O that used the major device number to determine the type of I/O, and block device drivers that needed to know if the device they controlled was the root or the swap device.

Drivers doing physical I/O now differentiate it either by using the block/character parameter of the combined open routine, or by marking the transfer in the *b_dev* field of the transfer's buffer. Drivers needing to know if they are the root device can find out using the following or something similar:

```
#include "sys/conf.h"
#include "sys/cmn_err.h"

extern struct bdevsw bdevsw[ ];

if ( bdevsw[major(rootdev)].d_open == xxopen ) {
  cmn_err(CE_CONT,"the xx driver is the root device\n");
        .
        .
        .
}
```

Drivers also should not and do not need to know the vector on which they interrupt. The underlying hardware determines the vectors on which a device is capable of interrupting. When the hardware is only capable of interrupting on one vector, there is little a driver writer can do beyond the timeout schemes discussed previously. If the vector is configurable on the card, some cards allow you to query the vector number directly.

Preconfigured drivers can simply check to see if someone else has already claimed that vector. Other drivers should encourage users to reconfigure when interrupts appear to get lost.

Using configurable port addresses poses similar issues. Like an advisory locking scheme, two drivers should usually be able to mitigate the port addresses and interrupt vectors between them, but a poorly written driver can cause problems for the whole system, sometimes making it look like some other driver is at fault.

Preparing Drivers to Use custom(C)

The best thing you can do for the end user is to supply a driver installation shell script for use with **custom**. With such a script, a user has only to type **custom** and select options from the menus.

The **custom** utility extracts the contents of your driver installation floppy, using them to control the custom installation procedure. **custom** requires the presence of the following:

- On each floppy volume, a product identification file whose name is derived from the driver package name, the volume, and a machine identification string
- The object module containing your device driver
- A *permlist*, or a file containing the file permissions for the other files and what volumes and packages they belong to.
- The driver installation shell script that forms the table entries binding driver and kernel.

All files on the driver installation floppy should be given by relative pathname, starting at the root. For example, if */bin/ls* were on the floppy, its name on the floppy should be *./bin/ls* .

The product identification file has a name of the following form:

```
./tmp/_lbl/prd=sidd/typ=386AT/rel=1.0.0/vol=01
```

where *sidd* is the driver's prefix (in this case, it stands for Sample Installable Device Driver), and 386AT is a machine-type specifier. To find the type specifier for your machine, check the file */etc/perms/inst* on your system. If you are developing for a different system, check the */etc/perms/inst* file on that system for the type identifier for that machine.

In the above example, 1.0.0 is the software release number of the driver, and 01 is the volume number of the floppy containing the driver. Note that there is no volume 0: volume numbers must start at 01 and be consecutive.

This file must exist on each volume of your driver installation set (incrementing the volume number). It can be an empty file; its contents are ignored.

The *permlist* is a file containing a list of the files on the floppy, their permissions, and their packages. It will be used by **custom** both as an argument to **fixperm**(ADM) and to determine which driver files belong to which package. This makes it easy for the user to install one driver in a driver suite containing many. The *permlist* must live in *./tmp/perms*. Below is a sample *permlist*:

```
:
#
#      Copyright (C) 1986-1989 The Santa Cruz Operation, Inc.
#      Copyright (C) 1986-1989 AT&T
#      Copyright (C) 1986-1989 Microsoft Corp.
#      All Rights Reserved.
#
#prd=sidd
#typ=386AT
#rel=1.0.0
#set="Sample Installable Device Driver"
#
# User id's:
#
uid    root          0
#
# Group id's:
#
gid    root          0
#
#
#!SIDD     11        Sample Installable Device Driver
#
# Fields are: package [d,f,x]mode, user/group, links,
# path, volume

SIDD    F644    root/root 1    ./tmp/perms/sidd                01
SIDD    F755    root/root 1    ./tmp/init.sidd                 01
SIDD    F644    root/root 1    ./etc/conf/pack.d/sidd/Driver.o 01
```

Some of the fields are self-explanatory and can be copied verbatim. The *prd*, *typ*, *rel*, and *set* fields are comments to **fixperm** but are meaningful to **custom**. They must agree with the *prd*, *typ*, and *rel* entries in the filename, above. The *set* field is used by custom when it prompts for the users choice of packages to install.

Fields starting with '#!' are package specifiers. At least one must be present so that **custom** has something to prompt for. The '11' in the #!SIDD field above is the size, in 512 byte blocks, (as reported by **du**(C)) of all the files in the package. The comment following the size is also used in **custom**(ADM) prompting.

The final section contains the package specifier, file type and permission, ownership, link count, file name and volume for each file on the distribution. The file type is **d** for directory, **x** for executable file, and **f** for normal file. If the file type is capitalized, the file is optional, and custom will not complain if it is missing. The files section is explained in more detail in **fixperm**(ADM).

Driver Installation Script Overview

A driver installation shell script should have the following duties:

- Check to see if the Link Kit is present, and install it if it isn't.
- Add the new driver entry points to the kernel using **configure**.
- Change directory to */etc/conf/cf.d*, and then run *link_unix* to link the kernel.
- Make the device nodes in */dev*.

Refer to the sample installation script provided in the */usr/lib/samples/scripts* directory in your software. The scripts provided there can be used as models for an installation script that you write.

Chapter 7

Writing a SCSI Driver

Introduction 7-1
- About This Chapter 7-2
- SCSI Devices 7-3
  - SCSI Device Configurations 7-3
- SCSI Commands 7-4

Driver Overview 7-5
- Host Adapter Communication With a Driver 7-5
- Driver Structures 7-6
  - Structure Initialization 7-6
  - Local Structures 7-6
  - Configuration Table 7-6

Request Block 7-8
- Request Block Format 7-8

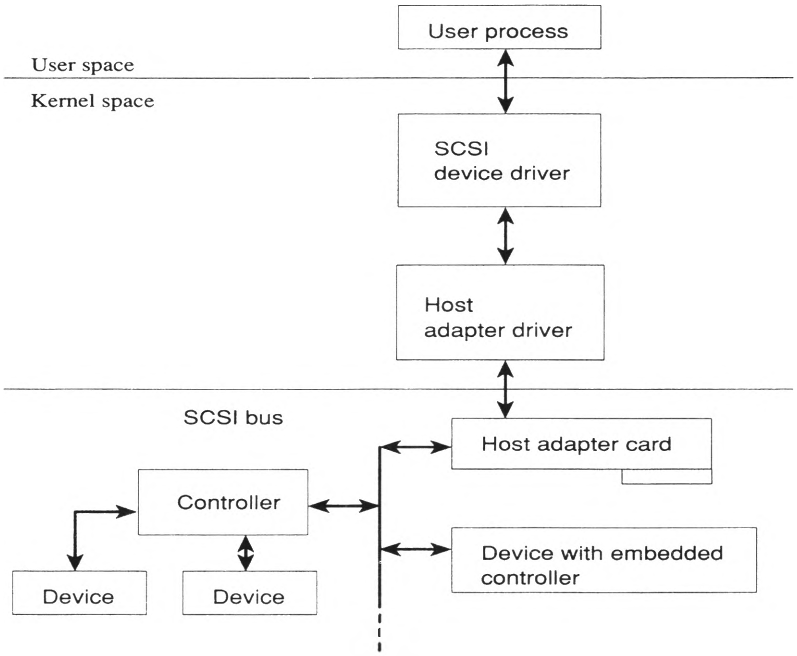
Writing a SCSI Device Driver 7-11
  - xxopen 7-12
  - xxioctl 7-13
  - xxbreakup 7-13
  - xxread and xxwrite 7-14
  - xxstrategy 7-15
  - xxstart 7-16
  - xxintr 7-17

Installing a SCSI Device or Host Adapter Driver 7-18
- Editing the mscsi File 7-19

Host Adapter Driver 7-20
- Initialization 7-20
- The xx_entry Routine 7-20
- I/O Control Command Routine 7-20
- Host Adapter Interrupt Routine 7-21

Introduction

This chapter describes how to write or maintain a driver for a device on the Small Computer Systems Interface (SCSI). The following diagram illustrates the interface:



The SCSI kernel routines are contained in the development system software, and provide a vehicle for creating or maintaining SCSI device drivers and host adapter drivers. A *host adapter* is a card that converts computer-independent SCSI communications protocol to computer-specific information that your computer can process. A *SCSI bus* is a cable connecting a host adapter to a series of SCSI controllers and their associated devices.

The SCSI software provided with your development system consists of a host adapter driver, a disk driver, a tape driver, and kernel routines that you can use when creating your SCSI device or host adapter drivers. The provided host adapter driver contains an optional I/O control command (ioctl) interface that permits pass-through control to the hardware on a SCSI bus. No additional software needs to be installed to start writing SCSI drivers.

SCSI is defined by the American National Standards Institute (ANSI), and the provided SCSI software is compatible with the ANSI X3.131-1986 and X3T9.2/85-52 Rev 4.B standards.

About This Chapter

This chapter covers the following topics:

- Introduction — An introduction to SCSI devices, device configurations, and SCSI commands.
- Driver Overview — Information about how the host adapter communicates with a driver, driver structures, structure initialization, local structures, and the configuration table.
- Request Blocks — A SCSI driver uses request blocks to convey commands and data to and from the device. This mechanism is the heart of a SCSI driver's interaction with a device.
- Driver Routines — A discussion of routines used in a SCSI device driver.
- Installing a SCSI Device Driver — A procedure for installing a device driver into the kernel.
- Host Adapter Driver — A discussion of the components of the host adapter driver and how it is installed into the kernel.

SCSI Devices

The small computer systems interface provides a computer with device independence within a *class* of devices. One class of device is disk drives, another class is tape drives, and so on. Using this interface, The following device classes can be added to a host computer without requiring modification to generic system hardware or software:

Device	Description
CD-ROM	SCSI cartridge disk, read-only memory
Communications	SCSI communications device
Disk	Either a SCSI "Bootable rootable" hard disk drive, or a SCSI disk drive
Media Changer	SCSI robot device for changing storage media
Optical	SCSI optical memory device
Processor	SCSI processor device
Scanner	SCSI scanner device
Tape	SCSI tape drive
WORM	SCSI Write-Once-Read-Many drive

The SCSI bus is a local I/O bus that can be operated at data rates of up to 5 megabytes per second.

The small computer systems interface uses logical rather than physical addressing for all data blocks. A logical unit may coincide with all or part of a peripheral device.

SCSI Device Configurations

Up to seven SCSI device controllers can be on a bus with each controller being able to address eight peripheral devices, making a possible total of up to 56 devices. With the two possible buses, the number can reach 112 devices. Multiple devices contending for the bus are handled by a priority system that awards interface control to the highest priority device.

Communication on the SCSI bus is only allowed between two devices at a time, the initiator and the target. The software provided with your system assigns the host adapter to be the initiator.

SCSI protocol provides for the connection of multiple initiators; that is, devices that can initiate an operation, and multiple targets. The supported arrangement is single-initiator, multiple-target.

SCSI Commands

A request to a peripheral device is performed by sending a command descriptor block (CDB) to the target. The CDB consists of an operation code byte, a logical unit number (LUN), command parameters, if any, and a control byte.

The operation code byte has a group code field and a command code field. There are eight different group codes and thirty-two different commands per group yielding 256 different operation codes.

Group codes are divided as follows:

- Group 0 - six-byte commands.
- Group 1 - ten-byte commands.
- Group 2 - reserved.
- Group 3 - reserved.
- Group 4 - reserved.
- Group 5 - twelve-byte commands.
- Group 6 - vendor unique.
- Group 7 - vendor unique.

Driver Overview

A SCSI driver communicates with the host adapter server instead of directly with the hardware. The host adapter driver contains normal device driver entry points. The SCSI device driver contains a subset of the normal driver routines; usually those routines required just for opening and closing the device, and for reading, writing, and processing information.

Each host and device driver has a unique major device number. Each driver is installed the same as any other driver, but the device driver information is also put in the */etc/conf/cf.d/mscsi* file. When a new kernel is built, the build software creates a configuration table from the information in *mscsi* that indexes each SCSI device to its respective controller and host adapter.

Host Adapter Communication With a Driver

The device driver communicates with the host adapter server via a request block. The request block contains the SCSI command block, relevant addresses and data lengths, and the interrupt function to call when the adapter server is finished with the request.

A device driver contains all routines necessary to operate a device. These include **xxopen, xxclose, xxread, xxwrite, xxstrategy, xxioctl, xxpoll, xxstart, xxintr,** and **xxhalt,** and **xxinit** driver routines. Guidelines for what each routine should include are described in Chapters 3 and 4 of this guide, and in the following sections if applicable.

A host adapter driver includes these routines: **xxinit, xxioctl,** and **xxintr**. The driver controls all initial setup of a host adapter, processes requests on behalf of other device drivers for access to SCSI devices, and allows user control of certain adapter-specific parameters.

Driver Structures

This section describes generic and local structures that are required in a SCSI driver.

Structure Initialization

At system initialization time, a device driver searches its configuration table and initializes all needed data structures. Information contained in the configuration tables accessed by the device driver includes the host adapter base address, the ID of the SCSI controller, and a device logical unit number (LUN).

A SCSI driver uses the configuration information to determine how the tunable parameters for a controller are set and how to work with these values.

Local Structures

If your driver is for a disk, you need an instance of the *diskinfo* structure defined in *sys/disk.h*. This structure is quite lengthy and should be studied in the header file. As with other disk drivers, you need an *xxtab* structure which is an instance of the *iobuf* structure. *xxtab* is described in Chapter 3, "Block Drivers."

Your driver should also have structures for maintaining drive and controller status, and a structure for disk sizes.

Configuration Table

A driver must provide one or more arrays for storing device information called the *configuration tables*. These tables are created automatically as described in the section entitled. "Installing a SCSI Driver." However, if your driver cannot conform to the naming convention established by the installation interface, you can create your own configuration tables. When creating your own configuration tables, you must also write your own version of the */etc/conf/bin/idscsi* program called by the Link Kit software. and you must inform users of your driver access information for your device. All of this extra overhead is provided in the Link Kit and should be used initially to ensure that your driver can be brought up and tested before you create your own system.

The SCSI configuration tables are used to map SCSI devices to each SCSI bus.

Configuration tables can be extended to provide additional features such as scatter-gather, linkable SCSI commands, zero latency reads, extended SCSI messages, and adapter target mode.

Configuration tables are defined as being device-specific or adapter-specific.

The device-specific configuration table includes information on SCSI ID and LUN (logical unit number) values, host adapter number, device name, and a pointer to the adapter entry function.

The adapter-specific configuration table contains the host adapter base address, the host adapter number, the interrupt and DMA channels, and a pointer to an adapter entry routine.

When the system is initialized configuration information is provided by the system based on data in the *mscsi* file of the Link Kit.

The configuration table is an instance of the *scsi_dev_cfg* structure defined in *sys/scsi.h* and must be terminated with 0xFF.
The fields of the *scsi_dev_cfg* structure are:

Field	**Description**
int index;	/* major number - filled in by kernel */
char *dev_name;	/* device prefix */
dev_t devnum;	/* device number */
unsigned char ha_num;	/* host adapter number (0, 1, 2, ...) */
unsigned char id;	/* SCSI priority and address */
unsigned char lun;	/* logical unit number of device */
int (*adapter_entry)( );	/* host adapter entry point */
struct exten *dext;	/* extensions to SCSI spec */

Each driver passes arguments to the host adapter driver via the *adapter_entry* entry point. This call is referred to elsewhere in this chapter as the **xx_entry** routine.

Request Block

Writing a SCSI driver is different from other drivers for a disk or tape unit in that a SCSI driver mainly builds information blocks that are sent to a device and then evaluated.

The SCSI device driver can only communicate with the SCSI hardware via the host adapter device driver. A request block is constructed with a request type. The only request type currently supported is REQ_SEND. All other request types are not supported and the request type values ranging from 1 to 50 are reserved for future use. Request types of 51 and above are undefined.

The logic for a routine in a SCSI driver centers on these steps:

1. Get a request block (a request block is described in the next section)
2. Populate the request block with command information
3. Send the request to the device
4. Examine the data that is returned.

Request Block Format

A request block is a structure used to communicate with the host adapter driver. This structure is named *scsi_io_req*, is also called REQ_IO, and is defined in *scsi.h*.

All fields in the request block can be written to except *host_sts*, host status, *target_sts*, target status, and *req_status*, the request status at interrupt time. All other field values must be set before sending the request to the host adapter driver. Some fields may not be implemented in all host adapter drivers. The request block structure contains information needed by a wide variety of host adapter drivers. As an example, the supported host adapter driver does not support linked commands at present. Thus the *link_ptr* and *link_id* fields are not accessed by the host adapter driver supplied with your software. Future support for the link command capability is possible; set these fields to 0 (zero) to ensure that conflicts do not occur.

The fields that are generic to any SCSI driver are:

Field	Description
unsigned short req_type;	/* one of the classes of calls to the adapter driver module */
struct scsi_io_req *req_forw;	/* forward pointer to next scsi_io request block */
struct exten *ext_p;	/* normally reserved, ptr to SCSI extensions structure */
char opcode;	/* operation code */
char id;	/* device ID */
char lun;	/* logical unit number */
char ha_num;	/* host adapter number */
char dir;	/* direction of transfer */
char cmdlen;	/* command length */
paddr_t data_len;	/* data length */
paddr_t data_ptr;	/* data pointer */
paddr_t data_blk;	/* logical block */
paddr_t link_ptr;	/* link pointer */
char link_id;	/* ID of linked block */
char host_sts;	/* host status */
char target_sts;	/* target status */
union scsi_cdb scsi_cmd;	/* command information */
char sense_len;	/* length of sense command */
paddr_t scsi_sense;	
paddr_t req_id;	/* request block ID */

Request Block

The following fields of the request block structure must be tailored for each driver type and are device-specific:

Field	Description
char use_flag;	/* zero if available */
char internal;	/* driver intrnl req if non-zero */
char jq;	/* job queue index */
char ctlr;	/* controller address */
char req_status;	/* req blk status in xxintr */
char adapter;	/* adapter base address */
char r_count;	/* retry count */
char hacmd;	/* host adapter command */
struct buf *rbuf;	/* request per this buf if non zero */
int (*io_intr)();	/* device driver intr handler */

typedef struct scsi_io_req REQ_IO;

The command information (*scsi_cdb*) member varies by the length of command and is defined as follows:

```
union scsi_cdb {
        struct SixCmd {
                unsigned char opcode;
                unsigned char misc:5;
                unsigned char lun:3;
                unsigned char data[3];
                unsigned char control;
        } six;
        struct TenCmd {
                unsigned char opcode;
                unsigned char misc:5;
                unsigned char lun:3;
                unsigned long block;
                unsigned char reserved;
                unsigned short length;
                unsigned char control;
        } ten;
        struct TwelveCmd {
                unsigned char opcode;
                unsigned char misc:5;
                unsigned char lun:3;
                unsigned long block;
                unsigned long length;
                unsigned char reserved;
                unsigned char control;
        } twelve;
        unsigned char raw[12+sizeof(struct scsi_sense)];
};
```

Writing a SCSI Device Driver

During the **xxopen** phase of the driver, errors and permissions are checked, and the device driver completes initialization.

The **xxread** and **xxwrite** routines call the **xxstrategy** routine directly or through the kernel's **physio**(K) routine.

The **xxstrategy** routine builds and queues a request taking care to check for possible errors. In addition, the request contains a pointer to an interrupt handler routine to call at interrupt time. This request is started by sending it to the host adapter driver via the configured **xx_entry** routine. The host adapter driver builds an adapter-specific request packet and sends it to the hardware.

At interrupt time, the host adapter driver handles the interrupt, determines the device that caused the interrupt, and passes the interrupt along with appropriate data structure pointers to the device driver's **xxintr** routine. This routine then processes the information and passes it back to the calling user process.

Generally, it is not necessary for a SCSI device driver to have an **xxinit** routine since most initialization tasks need only be done by the adapter driver. The SCSI device driver can get by with a "firsttime" flag in the **xxopen** routine. The advantage to this approach is that by open time, interrupts are enabled so that it is easier to handshake with the host adapter driver.

An **xxinit** routine should only be used for buffer management initialization, and data structure initialization. Talking to the hardware at **xxinit** time is not advised; do so during the first call to **xxopen**.

Each driver routine is described in the sections that follow. The logic information provided is based on the needs of a disk driver. Refer to Chapters 3 and 4 for more information about each routine in the context of a block driver or a character driver depending on your driver type. If you are writing a STREAMS driver, refer to Chapter 9. The SCSI driver routines follow.

xxopen

The **xxopen** routine opens the device for access.

Syntax:

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

int
xxopen(dev, rw, bc)
dev_t dev;
int rw, bc;
```

The logic is as follows:

1. Extract the minor device number. If the number exceeds the number of allowable devices, set *u.u_error* to ENXIO and return.

2. Call **scsi_getdev**(K) to get device information from the SCSI configuration table. If information is not found, set *u.u_error* to ENODEV and return.

3. Wake up each device served by your driver by sending a command and checking the resulting error codes.

4. Perform any additional setup tasks.

5. Exit the routine by releasing the last request block, turning off the flag to indicate that you are opening the device, waking up any sleeping processes waiting to open a device, and then calling **splx** to restore any previously set interrupts.

xxioctl

Establish conditional modules for each I/O control command.

Syntax:

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

int
xxioctl(dev, cmd, arg, mode)
dev_t dev;
int cmd, mode;
caddr_t arg;
```

The logic is as follows:

1. Call **scsi_getdev** to ensure that the *dev* argument is for a legitimate device. If not, set *u.u_error* to ENODEV and return. Then check that the device is present. If not, set *u.u_error* to ENODEV and return.

2. Provide for any other ioctls to fit your needs. If an improper command is specified, set *u.u_error* to EINVAL, and return.

xxbreakup

This routine breaks up a raw I/O request into smaller units for transfer to a direct memory access (DMA) controller. It contains only a call to **xxstrategy** in this driver.

---

xxread and xxwrite

These routines initiate raw I/O to or from a device.

Syntax:

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

int
xxread(dev)
dev_t dev;

int
xxwrite(dev)
dev_t dev;
```

These routines contain these logic steps:

1. Extract the minor device number for use by **physio**(K) when performing physical I/O.
2. Check the device number against the information in the configuration table. If it is improper, set *u.u_error* to ENODEV and return.
3. Get the size of the device from a local driver array of disk sizes.
4. Call **physck**(K) to ensure that the I/O is correct for the device size.
5. Call **physio**. This routine calls **xxbreakup** which then calls **xxstrategy** to perform the raw I/O.

---

xxstrategy

This routine schedules I/O and manages the buffer cache for driver I/O.

Syntax:

#include "sys/scsi.h"

int
xxstrategy(bp)
struct buf *bp;

The logic is as follows:

1. Check the configuration table for the device pointed to by *bp->b_dev*. If it is improper, set *u.u_error* to ENXIO and return.

2. Check *bp->b_blkno* and if less than zero, set *b_flags* to an error condition, call **iodone**(K) and return.

3. Check that the I/O request is reasonable. If not, set an error condition, call **iodone** and return.

4. Get an empty request block; sleep if necessary to obtain one.

5. Call **spl5** and then call **disksort**(K) to put the buffer pointed to by *bp* into the proper spot in the request queue.

6. Call **xxstart** to start the request on the queue for the logical unit. When done, restore the previously set **spl** level and return.

---

xxstart

This routine is used to take the passed request block and determine the exact track and sector for the read or write request and to then send the request to the device specified in the request block. Most SCSI devices handle bad track lockouts at the disk level, thus rendering driver-level checking obsolete.

Syntax:

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

int
xxstart(req_p)
REQ_IO *req_p;
```

The logic is:

1. Return immediately if the queue is empty or the device is busy.
2. Compute the proper track and sector coordinates and get the physical address from which data is being moved or to which data is being moved. Plug this information into the request block.
3. Complete building the request block for I/O and send it to the device.

---

xxintr

This routine handles interrupts. Interrupts raised by SCSI devices are caught by the host adapter which, in turn, calls the driver's interrupt routine via request block information. The interrupt routine is also used to wake up processes that are sleeping waiting for a free request block.

When the interrupt is finished, **xxstart** is called to process any remaining queued requests.

Syntax:

#include "sys/scsi.h"

int
xxintr(req_p)
REQ_IO *req_p;

1. Determine which device called the interrupt. Display a message on the console if the interrupt is spurious, and return.
2. Check the status flags for the host adapter and device. If an error occurred, put error information in the driver's *xxtab* structure. Otherwise, put I/O completion and status information in *xxtab*. In both cases, call **iodone** and then clear the request block.
3. Call **xxstart** to handle any outstanding requests.

Installing a SCSI Device or Host Adapter Driver

Installing a SCSI driver is performed as follows:

1. For SCSI device drivers (not host adapter drivers), the software provided with your system contains a system for installing drivers that simplifies creating configuration tables. SCSI depends on a series of configuration tables to access the proper device. To use this simplified system, your driver must be named with reserved names that reflect the type of device be used, and information must be added to the *mscsi* file described in this section. If this system is not workable for your needs, the configuration tables can be built manually with a custom example of the */etc/conf/bin/idscsi*. The possible names for drivers using the simplified system are:

Driver Name	**SCSI Device**
hd	"Bootable rootable" hard disk drive
Sdsk	Hard disk drive
Smed	Robot device for changing storage media
Sopt	Optical memory device
Spr	Processor device
Srom	Cartridge disk, read-only memory (CD-ROM)
Sscn	Scanner device
Stp	Tape drive
Stty	Communications device
Swrm	Write-once-read-many (WORM) drive

2. Again, just for device drivers, add an entry to the */etc/conf/cf.d/mscsi* file for each SCSI device. Instructions for editing this file are described in this section.

3. For both types of drivers, compile your driver into an object file using the **cc**(CP) commands shown in Chapter 6.

4. For both types of drivers, copy the driver object file to the */etc/conf/pack.d* directory.

5. For both types of drivers, add information about your driver into the */etc/conf/cf.d/mdevice* and */etc/conf/cf.d/sdevice*. Refer to the **mdevice**(F) and **sdevice**(F) manual pages for more information about the fields in each file.

6. For both types of drivers, create a new kernel with the **link_unix** program described in Chapter 6. All necessary tables required for SCSI are built by the programs called by **link_unix**.

Editing the mscsi File

Enter one entry for each device driver in the */etc/conf/cf.d/mscsi* file. These entries need not be grouped consecutively. This file has five fields with the information separated by either spaces or tabs. The fields are:

1	2	3	4	5
Host adapter driver prefix	*Supported device driver name*	*Host adapter number (0 or 1)*	*SCSI bus device ID number*	*SCSI bus device LUN*

The fields are:

Host adapter driver prefix
The two-, three-, or four-letter host adapter driver prefix name. This name is stored in the *mdevice* file associated with the host adapter driver. The prefix name given here must be the same as that in the *mdevice* file. Refer to the **mdevice**(F) manual page for more information.

Supported device driver name
The name of the device driver. The name must be one of the previously described SCSI driver names.

Host adapter number
The number of host adapter card(s) that will be installed on the system when it is in use. Two host adapters from the same vendor can be supported. The possible values are 0 (zero) or 1. The host adapter card(s) do not have to be present in the computer until the driver is actually being executed.

SCSI bus device ID number
The identification number for a specific controller. The possible values range from 0 (zero) to 7.

SCSI bus device LUN
The identification number for a specific device on a controller. The device number is commonly referred to as a logical unit number (LUN). The possible values range from 0 (zero) to 7.

Host Adapter Driver

The host adapter driver is a more complicated piece of software than the device driver because it must be specific enough to use the features of the hardware for which it was intended, yet be generic enough to be able to field as-yet-unwritten SCSI device driver requests.

Initialization

The host adapter driver may have an initialization routine to probe the hardware for configuration information, and do initial board setup. The host adapter driver does not normally have an **xxopen** routine.

The xx_entry Routine

The **xx_entry** host adapter entry routine is the common entry point for all SCSI device drivers. In this routine the driver writer must build his adapter-specific command blocks with information garnered from the SCSI request block. These command blocks, or whatever data structure is appropriate (maybe just bytes, or words), are then sent to the host adapter hardware.

Most device drivers will not queue their I/O requests. It is up to the host adapter driver writer to handle these requests as they come in, and perhaps to internally queue the requests. For example, the supported host adapter has an internal buffer that can handle at least 16 outstanding requests. The host adapter driver makes use of this fact to send requests as quickly as possible to the host adapter.

I/O Control Command Routine

The **xxioctl** routine allows the user to control some adapter tunable parameters, such as, bus-on/off time, and the number of adapter specific data structures.

Host Adapter Interrupt Routine

The host adapter interrupt routine handles the interrupt from the host adapter hardware, determines which device needs service, sets up the returned information in an appropriate data structure if needed, and calls the appropriate SCSI device interrupt routine. All SCSI device driver interrupt routines normally expect a pointer to a SCSI request block. Information such as data address, device and host adapter status, data length, and error status are contained in a request block.

The host adapter driver, after servicing a completion interrupt, determines the target SCSI unit and communicates back to the calling device driver via the **io_intr** routine specified in the *scsi_io_req* structure. The adapter passes a pointer to a *scsi_io_req* structure to communicate information back to the device.

Chapter 8

Line Disciplines

Introduction 8-1
    What Is a Line Discipline? 8-1
    Provided Line Disciplines 8-2
    Line Discipline Routines 8-4
    Writing a Line Discipline 8-6

tty Structure 8-7

The xxproc Routine 8-12

ttiocom Routine 8-16
        Syntax 8-16
    Description 8-17

Introduction

A line discipline is a set of routines that examine data received from a character device and take a pre-defined action when special characters are received. The special characters can be anything required for your driver. If your driver is a terminal handler, the special characters might be <Return>, <Break>, <Bksp>, or a control character. A line discipline is like a user level filter in that the input is translated to form an alternative output. However, unlike a filter, only one line discipline can be active at a time; line disciplines cannot be stacked together. If your driver requires modular, stackable line disciplines, STREAMS should be used. Refer to Chapter 9, "STREAMS" for more information.

The design of a line discipline is based on handling data from a terminal device in which text characters are combined with control characters, and other characters available on the keyboard. With the advent of international character sets, a line discipline can translate characters that are outside the ASCII character scope. Traditionally, the line discipline was designed when hardware was relatively limited and software had to handle a greater share of the character translation. Today, however, hardware controllers can all but eliminate the need for lengthy line disciplines, and software can be limited.

What Is a Line Discipline?

A line discipline consists of a number of routines that work together taking data from the terminal or other serial device, to process out the non-text characters and give it to the user program. Another set handles the output side of device interaction, taking characters from the user process, and handing them ultimately to the device output mechanism, such as the screen on a terminal. The interaction between the routines is handled by a series of five buffers or queues that are used to store data so that another routine can access it.

When line disciplines are included in a device driver, a user process can select among the installed line discipline packages using the **ioctl**(S) system call. Behavior of each line discipline package can be configured with additional **ioctl** calls. Although there are several line discipline packages available, try to use line discipline zero (0) to maintain a traditional tty user interface. Many of the details of line discipline zero are documented in the **termio**(M) and **tty**(K) manual pages.

Some hardware, such as ``smart'' controllers, may implement various character processing logic in firmware or hardware, but the driver writer should code the driver to maintain the ability to switch between line disciplines with user ioctls.

Typically, line disciplines provide the following capabilities:

- Keyboard input functions with character processing which includes editing, signals, and flow control
- Output display including character translations, expansions, and delays
- Buffering between task time requests and interrupt routines
- Disparity of device responses from user process requests
- Ability to coordinate driver timing and process control
- Implementation of a controlling terminal for process group control
- Posting of keyboard signals
- A standard programmable interface for user process configuration through ioctls
- Support for non-English character sets

Provided Line Disciplines

The line disciplines provided by your system are referred to by numbers starting with zero, with a maximum of up to 255. Three line disciplines are provided in the kernel. Line discipline zero is for terminal (tty) devices, line discipline one is for **shl**(C) shell layers. Line discipline two is used by the mouse driver.

A line discipline is an instance of the *linesw* structure defined in *sys/conf.h*. The line discipline switch table has the following fields:

Type	**Field**	**Description**
`int`	`(*l_open)();`	`/* Initialize a tty device */`
`int`	`(*l_close)();`	`/* Discontinue tty device access */`
`int`	`(*l_read)();`	`/* Copy data to a user process */`
`int`	`(*l_write)();`	`/* Copy user process data to kernel */`
`int`	`(*l_ioctl)();`	`/* Open, close, or change a line discipline */`
`int`	`(*l_input)();`	`/* Get input on receive data interrupt */`
`int`	`(*l_output)();`	`/* Send output to device on interrupt */`
`int`	`(*l_mdmint)();`	`/* Process modem-generated interrupt */`

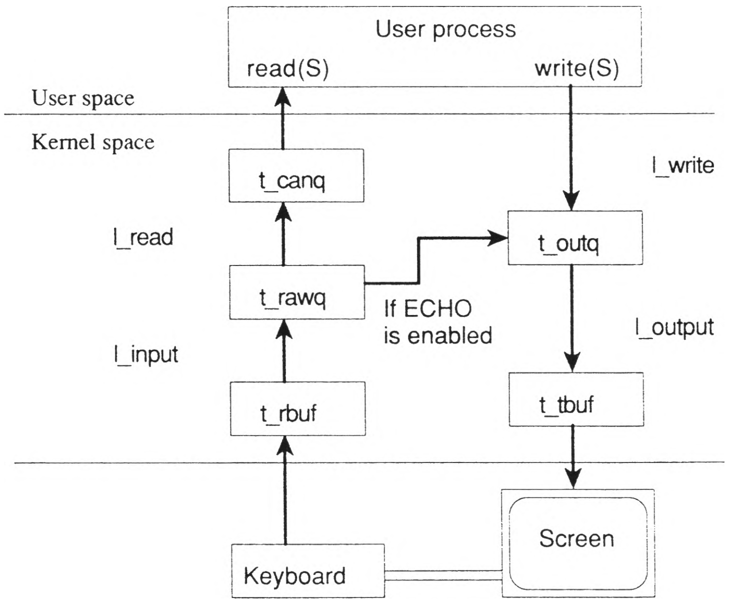
```
extern struct linesw linesw[];
```

In the kernel, all of the line disciplines are listed in the */etc/conf/pack.d/kernel/space.c* file. In this file, each routine in the line discipline is declared as external, listed in the **linesw** structure definition, and the total number of line disciplines is assigned to the **linecnt** variable. If a routine is not required for the driver, **nulldev** is coded into the position. When executed, **nulldev** returns a NULL to the calling process. The index into **linesw** is stored in the **t_line** member of the *tty* structure.

Getting a line discipline recognized by the kernel is easy by comparison to the designing, coding, testing, linking, and debugging required to produce it. Updating the *kernel/space.c* file and correctly installing the line discipline routines are the only requirements.

Line Discipline Routines

Each line discipline is a set of eight routines that provide the following device functions: open, close, read, write, I/O control (ioctl), receive data, transmit data, and modem interrupt. The routines are called through the **linesw** members, **l_open, l_close, l_read, l_write, l_ioctl, l_input, l_output,** and **l_mdmint**, respectively. The following picture depicts how the line discipline routines are used to move data between a device and a user program:



A typical call to a line discipline routine is:

```
(*linesw[tp->t_line].l_ioctl)(tp, LDOPEN, 0, mode);
```

Where `tp` is an instance of the *tty* structure associated with a device, `t_line` is the index value of which line discipline is being called, `l_ioctl` is the line discipline routine for I/O control commands, and `tp, LDOPEN, 0,` and `mode` are arguments passed to the **l_ioctl** routine.

Most line disciplines, including those provided in the kernel do not provide a modem interrupt handler. However, this functionality can be added if the need exists.

All provided line disciplines require an **xxproc** driver routine for handling special tasks required by the line disciplines. This routine is called by all of the routines in line discipline zero and contains code to handle the tasks required of it. Detailed information is provided in the "xxproc Routine" section of this chapter.

The line discipline members, the arguments with which each is called, and the driver routine from which each is called are shown as follows:

Line Discipline Routine	*Called By*	*Purpose*
l_open(*tp*)	**xxopen**	open a device
l_close(*tp*)	**xxclose**	close a device
l_read(*tp*)	**xxread**	convey processed data to user process
l_write(*tp*)	**xxwrite**	convey data from user process to the output queue
l_ioctl(*tp, cmd, arg, mode*)	†	device I/O control command
l_input(*tp, code*)	**xxintr**	get data from a device
l_output(*tp*)	**xxintr**	output data to a device
l_mdmint(*tp*)	—	called by **xxintr** to service an interrupt generated by a modem (not currently used)

† The driver **xxioctl** routine calls the **ttiocom**(K) routine which then calls the **l_ioctl** routine.

8

Writing a Line Discipline

The stages involved in writing a line discipline vary from using existing routines and having little or no choice over how data is interpreted to writing a full line discipline and having complete control. Either end of the line discipline spectrum is complicated and difficult to program.

Examining the */etc/conf/pack.d/kernel/space.c* file reveals the extent to which the provided line disciplines used references to the line discipline zero routines, put **nulldev** into unneeded positions, created custom routines, or mixed and matched.

The existing line disciplines are:

linesw Member	*t_line* Index Values		
	0 - tty driver	**1** - shell layers	**2** - mouse driver
l_open	**ttopen**	**nulldev**	**moopen**
l_close	**ttclose**	**nulldev**	**moclose**
l_read	**ttread**	**nulldev**	**moread**
l_write	**ttwrite**	**nulldev**	**ttwrite**
l_ioctl	**ttioctl**	**nulldev**	**moioctl**
l_input	**ttin**	**sxtin**	**moin**
l_output	**ttout**	**sxtout**	**ttout**
l_mdmint	**nulldev**	**nulldev**	**nulldev**

As you can see, the shell layers driver only needed routines for receiving data and transmitting it. All other functionality was nulled out. The mouse driver, required all of the line discipline functionality but was able to use the line discipline zero write and transmit data routines instead of writing new routines.

A complete description of the line discipline zero routines is provided on the **tty**(K) manual page.

tty Structure

The *tty* structure is the central point of a line discipline and is used by a line discipline to point to I/O buffers and to store flags, states, and conditions used when handling data from a tty device. *sys/tty.h* contains the definition for this structure. Refer to *tty.h* for descriptions of the *cblock, ccblock*, and *clist* structures used in the *tty* structure's description.

The fields in the *tty* structure are:

Type	**Field**	**Description**
clist	t_rawq;	/* Raw input queue */
struct clist	t_canq;	/* Canonical queue */
struct clist	t_outq;	/* Output queue */
struct ccblock	t_tbuf;	/* Transmit control block */
struct ccblock	t_rbuf;	/* Receive control block */
int	(* t_proc) ();	/* Routine for device functions */
ushort	t_iflag;	/* Input modes */
ushort	t_oflag;	/* Output modes */
ushort	t_cflag;	/* Control modes */
ushort	t_lflag;	/* Line discipline modes */
short	t_state;	/* Internal state */
short	t_pgrp;	/* Process group name */
char	t_line;	/* Line discipline index */
char	t_delct;	/* Delimiter count */
char	t_term;	/* Terminal type */
char	t_tmflag;	/* Terminal flags */
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+5];	/* Settable control chars */
unsigned char	t_schar;	/* Save timeout character */ /* instead of using t_lflag */

The following fields of this structure are significant for driver development:

- **t_rawq** points to the first cblock in the device's raw input character queue where data is placed after being received from the device by **ttin**(K). This field is read only.

- **t_canq** points to the first cblock in the device's processed character queue. The data placed here is processed depending on the setting of the ICANON flag following the conventions described in **termio**(M). Data is placed in this queue by the **canon**(K) routine. This field is read only.

- **t_outq** points to the first cblock in the device's output queue. Data is placed here by either **ttxput**(K) or **ttwrite**(K). Data is conveyed out of this queue by **ttout**(K) when a transmit data interrupt is serviced. This field is read only.

- **t_tbuf** points to the first character control block for a device's data transmit buffer. A character control block points to the data field in a cblock. Data is placed in this buffer by **ttout**(K) and taken out of the buffer by the terminal controller. The driver can set the fields in the ccblock.

- **t_rbuf** points to the first character control block for a device's data receive buffer. Data is placed into the receive buffer by the device-dependent input routine or from the device controller, and taken out by **ttin**(K). The driver can set the fields in the ccblock.

- **t_proc** is the name of the driver's **xxproc** routine. This field is settable by a driver and must be set to the name of the driver's **xxproc** routine.

- **t_iflag, t_oflag, t_cflag,** and **t_lflag** are mode flags. Refer to the descriptions in **termio**(M) and substitute "**c_**" for "**t_**" when referencing the descriptions. For example, **t_iflag** is described on the manual page as **c_iflag**. These fields are read only. Set values in the *termio* structure, and then use the TCSETA ioctl of **ttiocom**(K) to write to the flag fields of the *tty* structure. Similarly, use the TCGETA ioctl of **ttiocom** to move values from the *tty* structure into the *termio* structure for reading.

- **t_state** is the line discipline status field. Values can be ORed into this field. Use AND only when removing a previously set value; never clear **t_state**. Possible values are described following this list.

- **t_pgrp** the process group ID for the terminal device. This field is read only.
- **t_line** is the number of the current line discipline for the terminal device. This field is writable by the driver.
- The remaining fields are used internally in the line discipline zero routines and are read only. A description of **t_cc** can be found on the **termio**(M) manual page listed under **c_cc**.

The possible values for **t_state** are:

BUSY	Output is in progress
CARR_ON	Carrier is present
CLESC	Last processed character was <Escape>
EXTPROC	A peripheral device or routine is responsible for processing characters. In line discipline zero, EXTPROC is called on input in the **ttin** routine for processing by a KMC controller and in the **ttxput** routine for mapping characters for other international character sets on output.
IASLP	Processes are sleeping waiting for characters to appear in the raw input queue, **t_rawq**.
ISOPEN	The device is open and can be accessed. This flag must be disabled before calling the **l_close** routine to permit a device close to occur.
OASLP	Processes are sleeping waiting for characters to appear in the output queue, **t_outq**.
RTO	A timeout is in progress (**tttimeo** has been called) for a device operating in raw input mode where no canonical processing is taking place. This state occurs when the VTIME value is specified. Refer to the description on the **termio**(M) manual page.
TACT	A timeout is in progress for the device (**tttimeo** has been called to wait until characters are received on input).

TBLOCK	Transmission from the terminal is blocked. This state is used to control the backlog of input that's been read but not yet handled. If a user types more input than the system can handle before any process intervenes to read it in, then the input buffers are flushed. However, if a process capable of understanding <Ctrl>-s and <Ctrl>-q sequences is generating the input, TBLOCK is set in the driver to block transmission. The NOT state of TBLOCK is set to release transmission. When a driver's **xxproc** routine finds TBLOCK set, the driver must set **t_state** to BUSY and then use **outb**(K) to tell the controller to halt data transmission.
TIMEOUT	Indicates that a time out condition is in effect and that further processing should not occur. This condition is set by **ttywait**(K) to let the asynchronous receiver/transmitter (UART) drain of characters, and by the driver's **xxproc** routine when a <Break> character is received.
TTIOW	Processes are sleeping waiting for completion of output to the terminal. Processes setting this flag must sleep on the address of **t_oflag** at TTOPRI. Processes put to sleep using TTIOW are awaken by either **ttyflush** when flushing read buffers, or by **ttout** when no characters remain in the output queue, **t_outq**. This flag is cleared by some kernel drivers and should not be counted on for new development.
TTSTOP	Output has been stopped by a <Ctrl>-s character received from the terminal. This flag is set and cleared by the driver's **xxproc** routine only, and is generally set in the T_SUSPEND case.
TTXOFF	Input data has hit the high water mark and must be retrieved by a user process before more data is received. This flag is set and cleared by the driver's **xxproc** routine only. When this state occurs, the driver should use **outb**(K) to tell the controller to stop sending data. This flag is generally set in the **xxproc** routine's T_BLOCK case.

TTXON	Input data has hit the low water mark due to the data being received and processed. The device needs to send more data. This flag is set and cleared by the driver's **xxproc** routine only. When detected, the driver should use **outb** to tell the controller to send data. This flag is generally set in the **xxproc** routine's T_UNBLOCK case.
WOPEN	The driver is waiting for an open to complete. This flag is set in a driver, but cleared by **ttopen**(K) when a device is open.

The xxproc Routine

The driver's **xxproc** (procedure) routine is an integral part of the routines that comprise line discipline zero. If your driver uses **ttopen**(K), **ttclose**(K), **ttread**(K), **ttwrite**(K), **ttioctl**(K), **ttin**(K), or **ttout**(K) in line discipline zero, or **canon**(K), **ttiocom**(K), **ttrstrt**(K), or **ttyflush**(K), then you must have an **xxproc** routine to service the calls from the kernel routines.

The syntax for **xxproc** is:

```
int
xxproc(tp, command)
struct tty *tp;
int command;
```

Where *tp* is a pointer to a *tty* structure and *command* is a command name described in *sys/tty.h*. The logic shown below for each *command* choice is applicable to many types of drivers, even those that are accessing a device. A driver that doesn't access a device can still use the *tty* structure and its queue and pointer format.

Possible values for *command*, in alphabetic order, are:

T_BLOCK	Send a command to prohibit further input. T_BLOCK is generally called when the number of characters placed in **t_rawq** is greater than the high water mark. From line discipline routines, this command is called by **ttin**.
T_BREAK	Send a break signal to a tty device.
T_DISCONNECT	Send a command to a terminal controller requesting that carrier be dropped.
T_INPUT	Initiate input to a device.
T_OUTPUT	Initiate output to a device if it is not busy or suspended.

T_PARM	Change *tty* structure parameters.
T_RESUME	Resume terminal output because a <Ctrl-q> has been received.
T_RFLUSH	Send command to terminal controller to flush terminal input queue.
T_SUSPEND	Send command to terminal controller to suspend output because a <Ctrl-s> has been received.
T_SWTCH	Switch between **shl**(C) layers because the VSWTCH character has been received.
T_TIME	Delay timing corresponding to use of VTIME has completed.
T_UNBLOCK	Send a command to the terminal controller to start sending characters because the current input has fallen below the high water mark.
T_WFLUSH	Send a command to the terminal controller to flush the output queue.

An **xxproc** routine, in form, is like a driver's **xxioctl** routine in that **xxproc** is composed of a series of case statements switched on the *command* argument to the routine.

The xxproc Routine

In the following example of an **xxproc** routine from a serial device driver, the **xx_start** routine is a device-specific routine that sends commands to the I/O ports of the device.

```
xx_proc(tp, cmd)
register struct tty *tp;
{
        int ttrstrt();

        switch(cmd) {
        case T_SUSPEND:
                /* suspended with a ^S from the user */
                tp->t_state |= TTSTOP;
                break;
        case T_WFLUSH:
                if (tp->t_tbuf.c_ptr)
                    tp->t_tbuf.c_ptr -=
                        tp->t_tbuf.c_size -
                                tp->t_tbuf.c_count;
                (*linesw[tp->t_line].l_output)(tp);
                s=spl7();
                tp->t_state &= ~TTSTOP;
                xx_start(dev);
                splx(s);
                break;
        case T_RESUME:
                /* ^Q or other resume character */
                /* from the keyboard             */
                s=spl7();
                tp->t_state &= ~TTSTOP;
                xx_start(dev);
                splx(s);
                break;
        case T_TIME:
                s=spl7();
                tp->t_state &= ~TIMEOUT;
                xx_start(dev);
                splx(s);
                break;
        case T_RFLUSH:
                /* flush read queue.  If the process is
                 * blocked because it is nearing TTYHOG,
                 * send it ^Q to wake it up again in unblock.
                 * the rawq/canq flush is done separately.
                 */
                s = spl7();
                xx_start(dev);
                splx(s);
                if (( tp->t_state & TBLOCK ) == 0 )
                        break;
                /* fall through */
        case T_UNBLOCK:
                /* block and unblock are used to control
                 * the backlog of input that's been read,
                 * but not yet dispersed.  If a user types
                 * more input than the system can handle
                 * before any process steps up to read it
                 * in, the input is flushed.  However, if a
                 * process capable of understanding ^S/^Q
                 * sequences is generating the input, we use
                 * block and unblock to send ^S/^Q's out and
                 * control excess input.  cu is one utility
                 * that does this.  The input mode bit IXOFF
                 * determines whether or not the terminal
                 * should be fed ^S/^Q's.
                 */
                s=spl7();
                tp->t_state &= ~(TBLOCK|TTXOFF);
                tp->t_state |= TTXON;
                xx_start(dev);
                splx(s);
                break;
        case T_BLOCK:
                s=spl7();
                tp->t_state &= ~TTXON;
                tp->t_state |= (TBLOCK|TTXOFF);
                xx_start(dev);
                splx(s);
                break;
        case T_BREAK:
                tp->t_state |= TIMEOUT;
                timeout(ttrstrt,(caddr_t)tp,Hz/4);
                break;
        case T_PARM:
                xx_param(dev);
                break;
        default:
                break;
        }
}
```

ttiocom Routine

The **ttiocom**(K) routine is used by many user processes and the **xxioctl** routine to alter the state of the *tty* and *termio* structures. The information provided in this section is the same as that provided on the **ttiocom** manual page and is included here for ease of reference.

Syntax

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

int ttiocom(tp, cmd, arg, mode)
struct tty *tp;
int cmd, arg, mode;
```

Parameters

tp	Pointer to an instance of the *tty* structure for a tty device
cmd	I/O control command passed through from the user program
arg	argument to the I/O control command, also passed through from the user program
mode	indicates the mode by which the file was opened. The modes are assigned by the kernel and are interpreted into flag values that are defined in *sys/file.h*. Possible values are FNDELAY, FREAD, FSTOPIO, FWRITE.

Description

ttiocom sends an I/O control command to the tty device. Valid commands (the *cmd* argument to **ttiocom**) are:

- IOC_SELECT — determine if a character can be read from or written to a tty device without blocking (going to sleep in the process). *mode* can be SELREAD or SELWRITE. NOTE: IOC_SELECT must not be called from an interrupt routine and *sleep* must not be called just prior to calling this I/O control command. IOC_SELECT calls *ttselect.*

- IOCTYPE — return the name of the last I/O control command called. *u.u_rvall* is set to the value of TIOC. IOCTYPE must not be called from an interrupt routine.

- TCSETAF, TCSETAW, TCSETA, TCGETA, TCSBRK, TCXONC, TCFLSH — explained on the **termio**(M) manual page. TCSETAW and TCSETAF call *ttywait*. TCSETAF calls *ttyflush*. TCSETA calls *ttioctl* when opening a new line discipline and when changing the value of the line discipline flag, *t_lflag*. TCSBRK calls *ttywait.* TCXONC calls the driver *xxproc* routine with varying arguments depending on the *arg* argument to **ttiocom**. TCFLSH calls *ttyflush.* TCGETA sets *u.u_error* to EFAULT if a paging error occurs while trying to return the requested *tty* structure. TCXONC sets *u.u_error* to EINVAL if *arg* is not 0, 1, 2, or 3. TCFLSH sets *u.u_error* to EINVAL if *arg* is not 0, 1, or 2. TCSETA sets *u.u_error* to EFAULT if the *tty* structure cannot be set, or to EINVAL if the requested line discipline is less than zero or greater than the maximum. *xxproc* is called by TCXONC as follows:

arg value	*xxproc argument*
0 (zero)	T_SUSPEND
1	T_RESUME
2	T_BLOCK
3	T_UNBLOCK

- FIORDCHK — check to see if characters are waiting to be read. 1 is returned if characters are waiting in *t_canq*. If ICANON is set, it is also possible for 1 to be returned when characters are not in *t_canq*, but there are characters in *t_delct*. If there are no characters in *t_canq* and ICANON is not set, and if there are characters in *t_rawq*, 1 is returned. If none of the queues have characters, 0 (zero) is returned. FIORDCHK causes *ttrdchk* to be called.

- XCSETAW — wait for the universal asynchronous receiver/transmitter (UART) to empty (waits 11 bit times depending on the terminal's baud rate). XCSETAW is a POSIX **termio** extension.

- XCSETAF — wait until the UART empties and then flush all read and write buffers (calls *ttyflush*). XCSETAF is a POSIX **termio** extension.

- XCSETA — set terminal parameters from the *tty* structure specified by the *arg* argument to **ttiocom**. XCSETA is a POSIX **termio** extension.

- XCGETA — get terminal parameters from a terminal's *tty* structure and put into the *tty* structure specified by the *arg* argument to **ttiocom**.

Chapter 9

STREAMS

Overview 9-1
    STREAMS Overview 9-1

STREAMS Driver Overview 9-4
    STREAMS Driver Routines 9-4
    xxput Routine — Put Messages 9-5
    xxsrv Routine — Service Messages 9-6
    xxopen Routine — Open Device/Module 9-6
    xxclose Routine — Close the Device/Module 9-7

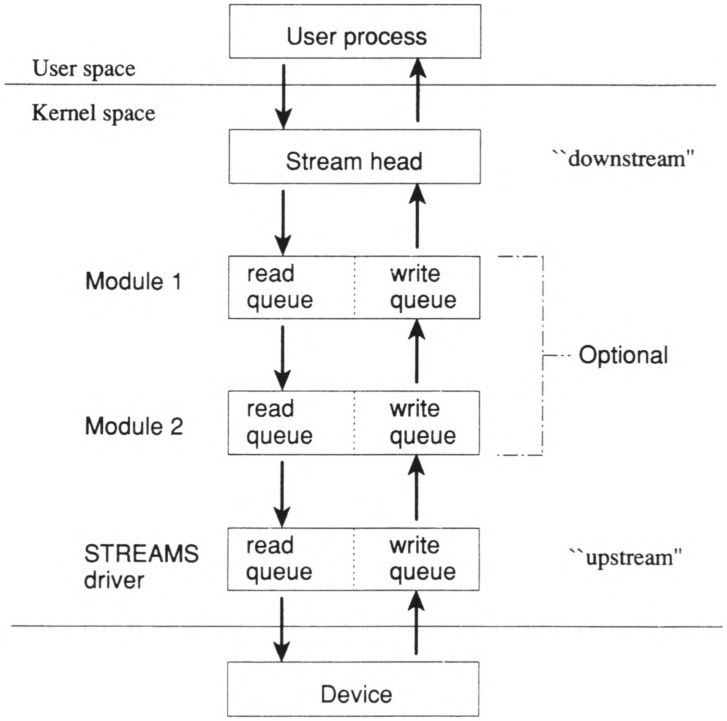
STREAMS Loop Back Driver 9-8

STREAMS Test Program 9-16

Overview

This chapter describes STREAMS and contains a sample STREAMS driver for a half-duplex pipe, described in terms of a loop back driver. At the end of the driver is a C program to test the driver.

The following diagram illustrates the parts of the STREAMS system:



STREAMS Overview

STREAMS was added to System V to improve the shortcomings of line disciplines. Line discipline routines require a great deal of interaction with all the component routines and are complex, difficult to write, and difficult to debug. In addition, only a single line discipline can be active at a time. STREAMS is designed so that you can create modular building blocks to connect a wide variety of hardware and software configurations.

Overview

STREAMS offers a system of linked lists of kernel data structures that combined with special routines to access the structures provide a full-duplex data path between a user process and a device. Each linked list of structures is referred to as a *Stream*. A Stream is comprised of at least one *STREAMS driver* and an interface to the user process, called the Stream *head*. A Stream may also include one or more pushable *modules* that a user can add or remove from the linked list. A module in its simplest form is two *queues*. A queue is an instance of the *queue_t* data structure. Unlike a driver, a module does not have an associated device file, and does not have user context. A Stream head is an interface provided for you in the STREAMS system and is standardized. Refer to the *STREAMS System* manual for more information.

The *queue_t* data structure, referenced on line 84 of the example driver and described in *sys/stream.h*, is the central point of a Stream. A STREAMS module and driver contains two *queue_t* structures, one for each data flow direction. A *queue_t* structure contains pointers to the members of the linked list.

The *queue_t* structure is as follows:

Type	**Field**	**Description**	
struct qinit	*q_qinfo;	/* procs and limits for queue	*/
struct msgb	*q_first;	/* first data block	*/
struct msgb	*q_last;	/* last data block	*/
struct queue	*q_next;	/* queue of next stream	*/
struct queue	*q_link;	/* to next queue for scheduling	*/
caddr_t	q_ptr;	/* to private data structure	*/
ushort	q_count;	/* number of blocks on queue	*/
ushort	q_flag;	/* queue state	*/
short	q_minpsz;	/* min packet size accepted by this module	*/
short	q_maxpsz;	/* max packet size accepted by this module	*/
ushort	q_hiwat;	/* queue high water mark	*/
ushort	q_lowat;	/* queue low water mark	*/

```
typedef struct queue queue_t;
```

A user process builds a Stream to meet its needs by pushing and/or removing modules from a Stream using the **ioctl**(S) system call. This is demonstrated in the sample driver in the last section of this chapter starting in line 45.

STREAMS programming is based on a flow of information up and down a Stream, that taken together is called a *message*. Transferred data, control information, queue commands, and errors and signals are some of the many messages that can be sent through a Stream. Messages are defined in the *sys/stream.h* header file as a series of **#defines**. Messages are passed between modules with the **put** routines described later in this chapter. A message is composed of one or more message blocks. A message block is a linked three-way unit consisting of the *msgb* and *datab* structures, and a variable-length buffer block. The message block structure, *msgb* provides a uniform mechanism for message exchange in a Stream.

The *msgb* structure, also called *mblk_t* is shown as follows:

Type	**Field**	**Description**	
struct msgb	*b_next;	/* next message on queue	*/
struct msgb	*b_prev;	/* previous message on queue	*/
struct msgb	*b_cont;	/* next message block on queue	*/
unsigned char	*b_rptr;	/* first unread byte in buffer	*/
unsigned char	*b_wptr;	/* first unwritten byte in buffer	*/
struct datab	*b_datap;	/* data block pointer	*/

```
typedef struct msgb mblk_t;
```

The data block structure contains more information to describe each message. The data block structure, *datab*, also called *dblk_t*, includes fields that describe a message and the number of messages pointing to the data block.

The structure of the data block is as follows:

Type	**Field**	**Description**	
struct datab	*db_freep;	/* used internally	*/
unsigned char	*db_base;	/* first byte of buffer	*/
unsigned char	*db_lim;	/* last byte +1 of buffer	*/
unsigned char	db_ref;	/* count of messages	*/
		/* pointing to this block	*/
unsigned char	db_type;	/* message type	*/
unsigned char	db_class;	/* used internally	*/

```
typedef struct datab dblk_t;
```

STREAMS Driver Overview

A STREAMS driver is conceptually cleaner in design than a comparable tty driver because the line discipline code is built into one or more modules. This frees the driver from having intricate input and output routines, thus focusing its functionality on its design and not that of character translation.

A driver requires three structures:

- *module_info* — Describes the specifics of a module. This structure is shown on line 40 of the sample driver.

- *qinit* — Describes what routines and structures comprise each side of a full-duplex Stream. An instance of the structure is required for each *queue*. Two *qinit* structures are used in the example driver and are referenced on lines 53 and 63. The routines pointed to by *qinit* are: **xxput, xxsrv, xxopen,** and **xxclose**. These are described in the next section.

- *streamtab* — Describes the names of the *qinit* structures and two NULL pointers for future use of multiplexing drivers. The *streamtab* structure described on line 72, is an entry in the character driver switch table (*cdevsw*). The *cdevsw* is defined in the *sys/conf.h* header file and provides entry points for kernel access of a character driver. A STREAMS driver is so indicated by the placement of a non-NULL value in the *d_str* field of the *cdevsw*. This tells the kernel to determine a STREAMS entry point routine from the *streamtab* definition in the driver.

STREAMS Driver Routines

Because STREAMS drivers use different entry points than other character drivers, the STREAMS driver routines differ to meet the needs of accessing the equally different structures and facilities.

The syntax and description for each of the STREAMS driver routines is shown in the listings that follow. Refer to the *STREAMS System* manual for more information.

xxput Routine — Put Messages

Syntax:

int
xxput(qp, mp)
queue_t *qp;
mblk_t *mp;

Where *qp* is a pointer to an instance of the *queue* structure, and *mp* is a pointer to an instance of the *msgb* structure.

The **xxput** routine has the following functions:

- Passing messages either up or down*stream* (the Stream head is the highest point upstream and the driver, the lowest point downstream)
- Deleting messages if required by the command
- Error detection

The **xxput** routine follows the format of an **xxioctl** routine in that it is a series of case conditions, switched on the type of message being sent:

```
        switch (mp->b_datap->db_type) { ...
```

Two **xxput** routines are required for a Stream, one for the read queue, called **xxrput**, and the other for writing, called **xxwput**. Requests handled by the respective **xxput** routines happen immediately; no scheduling is performed. A message is moved along the Stream queues by the head/module/driver calling the **xxput** routine of the head/module/driver in the direction of desired flow with the **putnext** macro described in the *STREAMS System* manual.

xxsrv Routine — Service Messages

Syntax:

```
int
xxsrv(queue-ptr)
queue_t *queue-ptr;
```

Where *queue-ptr* is an instance of the *queue* structure. The **xxsrv** routine has these functions:

- Retrieve messages
- Pass messages up or down the Stream

The **xxsrv** routine is shown in line 156 of the sample driver. This routine schedules the message interaction. Otherwise, it is identical in nature to an **xxput** routine.

xxopen Routine — Open Device/Module

Syntax:

```
int
xxopen(queue-ptr, dev, flag, sflag)
queue_t *queue-ptr;
dev_t dev;
int flag, sflag;
```

Where *queue-ptr* is a pointer to an instance of the *queue_t* structure, *dev* is the device number, *flag* is the same as the regular **xxopen** routine described in Chapter 4, and *sflag* is the Stream open flag.

The values of *sflag* can be:

- 0 (zero) for a normal driver open
- MODOPEN to open a module
- CLONEOPEN to indicate that the minor device number must be sought in an array implemented in your driver. Refer to lines 109 through 113 in the sample driver for more information on how a minor device number is selected. Also refer to the *STREAMS System* manual for more information on cloning. The sample driver sets *exst_lo* as an instance of *exst* on line 92. *devcnt* is the maximum number of minor numbers for a STREAMS device. CLONEOPEN is a feature of a system of dynamically allocate device nodes. Rather than having a series of pre-allocated device nodes taking system overhead, STREAMS permits device nodes to be implemented as needed. The **clone** driver (may not be implemented on all systems, contact your system administrator for details) creates inodes to meet the needs of the minor device being requested. These inodes do not have an associated */dev* file, they are only accessible via STREAMS.

An **xxopen** routine must return a number greater than or equal to zero if the open is successful, or OPENFAIL if not (as shown in line 116 of the sample driver). This routine has user context, can sleep, but only below PZERO, and must handle a signal internally instead of just returning.

xxclose Routine — Close the Device/Module

Syntax:

xxclose(queue-ptr)
queue_t *queue-ptr;

Where *queue-ptr* is an instance of the *queue* structure. **xxclose** closes access to the device or module.

STREAMS Loop Back Driver

The following table lists the line numbers of the driver routines:

Routine	*Line*	*Description*
exstopen(q, dev, flag, sflag)	103	open pseudo-device
exstclose(q)	135	close device
exstsrv(q)	156	service device queue
exstioctl(q, bp)	211	I/O control commands

The following routines in the loop back driver are described in the *STREAMS System* manual:

Routine	*Line*	*Description*
allocb	226	Allocate a message block
canput	167	Test for room in a queue
flushq	148	Flush a queue
freeb	262	Free a message block
freemsg	191	Free all message blocks in a message
getq	165	Get a message from a queue
putbq	168	Return a message to the beginning of a queue
putctl	289	Put a control message
putq	64	Put a message on a queue
qreply	188	Send a message on a stream in the reverse direction
RD	167	Get pointer to the read queue
unlinkb	261	Remove a message block from the head of a message
WR	128	Get pointer to the write queue

"Line" is the first occurrence of the routine.

```
/*
 * Copyright (C) 1989 The Santa Cruz Operation, Inc.
 *
 * The following code is a working example of a
 * STREAMS loop back driver. This driver illustrates
 * functions, structures, and utilities that are
 * described in detail in STREAMS System manual.
 *
 * This is a pseudo-driver designed to loop data from
 * one open Stream to another open Stream.  The user
 * processes view the associated files as a
 * half-duplex pipe.  This driver is a simple
 * multiplexer which passes messages from one
 * STREAMS' write queue to the same STREAMS' read
 * queue.  This driver also illustrates a STREAMS'
 * ioctl() function.
 *
 *     This driver does not include:
 *
 *     - an interrupt routine
 */

/* necessary include files */

#include "sys/types.h"
#include "sys/param.h"
#include "sys/sysmacros.h"
#include "sys/errno.h"
#include "sys/stream.h"
#include "sys/stropts.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/page.h"
#include "sys/seg.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/strlog.h"
#include "sys/log.h"


/* function declarations */

int nodev(), exstopen(), exstclose(), exstsrv();

/* streams structure declarations */

static struct module_info exstm_info = {
        40,         /* module ID number */
        "exst",     /* module name      */
        0,          /* min packet       */
        256,        /* max packet       */
        512,        /* hi-water mark    */
        256         /* lo-water mark    */
};
```

```
/*
   Service procedure structure for the read module.
   There is not a putq because the write procedure
   loops it back.
*/

static struct qinit exstrinit = {
        NULL,          /* put procedure                */
        exstsrv,       /* service procedure            */
        exstopen,      /* called on each open or push */
        exstclose,     /* called on last close         */
        NULL,          /* reserved for future use      */
        &exstm_info,   /* information structure        */
        NULL           /* statistics structure         */
};

/* Service procedure structure for the write module */

static struct qinit exstwinit = {
        putq,          /* put procedure                */
        exstsrv,       /* service procedure            */
        exstopen,      /* called on each open or push */
        exstclose,     /* called on last close         */
        NULL,          /* reserved for future use      */
        &exstm_info,   /* information structure        */
        NULL           /* statistics structure         */
};

struct streamtab exstinfo = {
        &exstrinit,    /* defines read queue           */
        &exstwinit,    /* defines write queue          */
        NULL,          /* for multiplexing drivers     */
        NULL           /* for multiplexing drivers     */
};

/*
   Private data structure, one per minor device number
   Used to create a private array of stream blocks.
*/

struct exst {
   unsigned exst_state; /* driver state flag, see below */
   queue_t  *exst_rdq;  /* queue pointer */
};

/* Driver state values.  */

#define EXSTOPEN 01      /* device is opened */
#define EXSTFAIL 02      /* open failed */
#define EXSTWOFF 04
#define NEXST     4      /* number of stream blocks */

/* Allocate streams blocks */

struct exst exst_lo[NEXST];
```

```
/* used to check the number of stream blocks */

int exstcnt = NEXST;

/* Loop back driver ioctl requests */

#define I_NOARG        20
#define I_INTARG       21
#define I_ERRNAK       23
#define I_ERROR        25
#define EXSTSLPTEST    32
#define I_SETHANG      42
#define I_SETERR       43

exstopen(q, dev, flag, sflag)
queue_t *q;
{
   struct exst *lp;

   dev = minor(dev);

   /* If CLONEOPEN, pick a minor device number to use.  */

   if (sflag == CLONEOPEN) {
      for (dev = 0; dev < exstcnt; dev++)
         if (!(exst_lo[dev].exst_state & EXSTOPEN))
            break;
   }

   /* check to see if we have a good device number */

   if ((dev < 0) || (dev >= exstcnt))
      return(OPENFAIL);          /* default = ENXIO */

   lp = &exst_lo[dev];

   /* check state */

   if (lp->exst_state & EXSTFAIL) {
      /* clear fail flag so it can be reopened later */
      lp->exst_state &= ~EXSTFAIL;
      return(OPENFAIL);
   }

   /* Setup data structures */

   if (!(lp->exst_state & EXSTOPEN)) {
      lp->exst_rdq = q;
      q->q_ptr = (caddr_t)lp;
      WR(q)->q_ptr = (caddr_t)lp;
      return(dev);
   }
   else
      if (q != lp->exst_rdq)
         return(OPENFAIL);  /* only one stream at a time! */

}
```

```
exstclose(q)
queue_t *q;
{
   /*
      If we are connected to a stream, break the
      linkage, and send a hang up message.
      The hangup message causes the stream head to fail
      writes, allow the queued data to be read completely,
      and then return EOF on subsequent reads
   */

   ((struct exst *)(q->q_ptr))->exst_state &=
                 ~(EXSTOPEN | EXSTWOFF);
   ((struct exst *)(q->q_ptr))->exst_rdq = NULL;
   flushq(WR(q), 1);
   q->q_ptr = NULL;

}


/*
 * Service routine takes messages off write queue and
 * sends them back up the read queue, processing them
 * along the way.
 */

exstsrv(q)
queue_t *q;
{
   mblk_t *bp;

   /* if exstsrv called from read */
   /* side set q to write side    */

   q = ((q)->q_flag&QREADR ? WR(q) : q);

   /* if upstream queue full, process */
   /* only priority messages          */

   while ((bp = getq(q)) != NULL) {
      if ((bp->b_datap->db_type) < QPCTL
                && !canput(RD(q)->q_next) ) {
         putbq(q, bp);
         return;
      }

      switch(bp->b_datap->db_type) {
         case M_IOCTL:
            exstioctl(q, bp);
               if (((struct exst *)(q->q_ptr))->exst_state
                        & EXSTSLPTEST)
                  return;
         break;
```

```
          /* if testing offset, calculate and */
          /* place at start of data message.  */

          case M_DATA:
          if (((struct exst *)(q->q_ptr))->exst_state
                & EXSTWOFF)
           (*(short *)(bp->b_rptr)) =
                (short)(bp->b_rptr - bp->b_datap->db_base);

          /* flow through */

          case M_PROTO:

          case M_PCPROTO:
             qreply(q, bp);
          break;

          case M_CTL:
            freemsg(bp);
          break;

          case M_FLUSH:
             if (*bp->b_rptr & FLUSHW) {
                flushq(q, FLUSHALL);
                *bp->b_rptr &= ~FLUSHW;
             }
             if (*bp->b_rptr & FLUSHR)
                qreply(q,bp);
             else freemsg(bp);
          break;

          default:
             freemsg(bp);
          break;
       }
    }
}

/*
   This routine tests the error ioctl commands
*/

exstioctl(q, bp)
queue_t *q;
mblk_t *bp;
{
   register s;
   int i, n;

   mblk_t *tmp;
   struct iocblk *iocbp;
   struct stroptions *so;
```

```
/* test for no offset in ioctl */

if (((struct exst *)(q->q_ptr))->exst_state & EXSTWOFF)
&& (bp->b_rptr != bp->b_datap->db_base)) {
   bp->b_datap->db_type = M_IOCNAK;
   qreply(q, bp);
   ((struct exst *)(q->q_ptr))->exst_state &= ~EXSTWOFF;
      if ((bp=allocb(sizeof(struct stroptions))) ==
                       NULL) {
         printf("exstsrv couldn't allocate data block\n");
         return;
      }
   bp->b_datap->db_type = M_SETOPTS;
   so = (struct stroptions *)bp->b_rptr;
   bp->b_wptr += sizeof(struct stroptions);
   so->so_flags = SO_WROFF;
   so->so_wroff = 0;
   qreply(q, bp);
   return;
}


/*
   Each particular ioctl has a special function
   for testing the STREAMS error mechanism.
*/

      iocbp = (struct iocblk *)bp->b_rptr;

      switch(iocbp->ioc_cmd) {

        case I_NOARG:
               bp->b_datap->db_type = M_IOCACK;
               qreply(q, bp);
               return;

        case I_INTARG:
               /*
                * Send integer argument back as return
                * value
                */
               if (bp->b_cont == NULL) {
                        freemsg(bp);
                        return;
               }
               iocbp->ioc_rval =
                        *((int *)bp->b_cont->b_rptr);
               tmp = unlinkb(bp);
               freeb(tmp);
               iocbp->ioc_count = 0;
               bp->b_datap->db_type = M_IOCACK;
               qreply(q, bp);
               return;
```

```
            case I_ERROR:
                  /*
                   * Verify that error return works.
                   */
                  iocbp->ioc_error = EPERM;
                  bp->b_datap->db_type = M_IOCACK;
                  qreply(q, bp);
                  return;

            case I_ERRNAK:
                  /*
                   * Send a NAK back with an error value.
                   */
                  iocbp->ioc_error = EPERM;
                  bp->b_datap->db_type = M_IOCNAK;
                  qreply(q, bp);
                  return;

            case I_SETHANG:
                  /*
                   * Send ACK followed by M_HANGUP upstream.
                   */
                  bp->b_datap->db_type = M_IOCACK;
                  qreply(q, bp);
                  putctl(RD(q)->q_next, M_HANGUP);
                  return;

            case I_SETERR:
                  /*
                   * Send ACK followed by M_ERROR upstream.
                   * Value is sent in second message block.
                   */
                  tmp = unlinkb(bp);
                  bp->b_datap->db_type = M_IOCACK;
                  ((struct iocblk *)bp->b_rptr)->ioc_count =
                                   0;
                  qreply(q, bp);
                  tmp->b_datap->db_type = M_ERROR;
                  qreply(q, tmp);
                  return;

            default:
                  /*
                   * NAK anything else.
                   */
                  bp->b_datap->db_type = M_IOCNAK;
                  qreply(q, bp);
                  return;
          }
}
```

STREAMS Test Program

```
/*

        Copyright (C) 1989 The Santa Cruz Operation, Inc.

*/

/* standard include files */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/stropts.h>

/* Loop back driver ioctl() commands */

#define I_NOARG     20  /* no argument test        */
#define I_INTARG    21  /* return integer test     */
#define I_ERRNAK    23  /* send NAK with error     */
#define I_ERROR     25  /* error return test       */
#define I_SETHANG   42  /* ACK w/upstream M_HANGUP */
#define I_SETERR    43  /* ACK w/upstream M_ERROR  */

/* global ioctl command structure */

struct strioctl ioc;

main()
{
   int fd, i;
   char buf[BUFSIZ];

   /*
      Open the loop back device. Note defined as /dev/cxst
      when configured into the kernel.

   */

   if ((fd=open("/dev/exst", O_RDWR, 0777)) == -1) {
      perror("open failed\n");
      exit(1);
   }
```

```
    /* Try writing and reading to the loop back device */

    printf("Enter a string to write to ");
    printf("the STREAM's loop back device\n");
    if ((fgets(buf,BUFSIZ,stdin)) == NULL) {
       perror("fgets failed");
       exit(1);
    }
    /*
     * function calls to read and
     * write to the loop back device
     */
    strwrite(fd,buf);
    strread(fd);

    /* Test the ioctl calls */

    strioctl(fd,I_NOARG,0,0,NULL);
    strwrite(fd,buf);
    strread(fd);

    strioctl(fd,I_INTARG,30,0,NULL);
    strwrite(fd,buf);
    strread(fd);

    strioctl(fd,I_ERRNAK,0,0,NULL);
    strwrite(fd,buf);
    strread(fd);

    strioctl(fd,I_ERROR,0,0,NULL);
    strwrite(fd,buf);
    strread(fd);

    strioctl(fd,I_SETHANG,0,0,NULL);
    strwrite(fd,buf);
    strread(fd);

    strioctl(fd,I_SETERR,0,0,NULL);
    strwrite(fd,buf);
    strread(fd);

    close(fd);

    exit(0);
}

strwrite(fd, s)
int fd;
char *s;
{
    int count;
    /* Try writing to the loop back device */

    printf("\nWriting to loop back device");
    if ((count = write(fd,s,BUFSIZ)) < 0) {
        perror("\nStream write failed");
        exit(2);
    }
}

strread(fd)
int fd;
{
    char buf[BUFSIZ];

    /* Try reading the loop back device */

    if ((read(fd,buf,BUFSIZ)) < 0) {
        perror("Stream read failed");
        exit(3);
    }
    /* print the read */

    printf("\nReading from loop back device\n");
    printf("String = %s\n",buf);

}

strioctl(fd,arg,time,len,s)
int fd;
int arg;
int time;
int len;
char *s;
{
    int i;
    char *p;

    ioc.ic_cmd = arg;
    ioc.ic_timeout = time;
    ioc.ic_len = len;
    ioc.ic_dp = s;

    switch (arg) {
        case I_NOARG : p = "i_noarg";
        break;
        case I_ERRNAK : p = "i_errnak";
        break;
        case I_ERROR : p = "i_error";
        break;
        case I_SETHANG : p = "i_sethang";
        break;
        case I_SETERR : p = "i_seterr";
        break;
        default :  p = "unknown ioctl";
        break;
        }
        printf("\nTrying ioctl call %s # %d\n",p,arg);
        if ((i = ioctl(fd,I_STR,&ioc)) < 0) {
            perror("ioctl failed");
            printf("return code = %d ioctl cmd = %d\n",i,arg);
        }

        printf("ioctl() return code = %d\n",i);
        sleep(3);
    }
```

Appendix A

Migrating XENIX Drivers to the System V Operating System

Introduction A-1

Binaries A-2

Change in the Device Number A-3

Changes in Structure Definitions A-4
- Structure Header Files A-4
- Changes to the buf Structure A-4
- user Structure A-6
- cblock Structure A-6
- devbuf Structure A-7

Changes in Kernel Routines A-8
- physio(K) A-8
- disksort(K) A-9
- xxopen, xxclose, xxread, xxwrite, xxioctl A-9
- major(K), minor(K) Device Number Macros A-9
- cmn_err(K) A-10
- sptalloc(K) and sptfree(K) A-11

Obsolete Kernel Routines A-12

General Notes A-14
- Halt Routines and Interrupts A-14
- Include Files A-14
- cram.h A-14
- Near and Far Keywords A-14
- xxprint Driver Routine A-15
- machdep.h A-15
- Compiler Directives A-15
- #ifdef M_S_UNIX A-15
- disk.c A-16
- Interrupt Vectors A-16
- Possible Problem Areas A-16

Converting IDDs A-17
- Overview A-17
- XENIX Files Relocated Under System V A-17
- Summary of XENIX vs. System V Directory Structures A-18
- Converting Installation Scripts A-19

Introduction

This appendix outlines the substantive changes that must be made in all XENIX version 2.3 drivers to allow them to be compiled under the System V release 3.2 environment. The necessity of some changes is decided by the CPU type. It is indicated in the text where this is the case.

Since the System V environment is different from the XENIX environment, read this document closely and note necessary changes to your drivers. There may be more changes required for some drivers than for others. As much as possible has been preserved of the old XENIX interface for reasons of backwards compatibility. By making the changes described in this paper, you should be able to bring your drivers to full System V 3.2 compatibility.

Because System V release 3.2 contains the merged XENIX product, the information presented below will contain references to this product where necessary.

This document includes information provided by Microsoft for device driver conversion from Microsoft XENIX drivers to the merged System V drivers.

This appendix describes the changes required to bring the driver code in line with the merged System V code standard. System V release 3.2 handles these changes and preserves older code with the pre-processor directives **#ifdef** and **#ifndef** and the conditional compiling of the code that these statements imply. Under System V release 3.2, two source code syntaxes are accepted: the syntax of XENIX release 2.3 and the syntax of System V.

Binaries

Under System V release 3.2, the kernel always runs 386 binaries.

In System V, both **COFF** and **x.out** 286 binaries are supported by the */bin/i286emul* and */bin/x286emul* user-level emulators. *i286emul* and *x286emul* trap system calls issued by a 286 program and either handle the system calls internally or perform the necessary argument conversions before issuing a 386 system call. Therefore, the device driver code that was used to support system calls from a 286 binary is no longer necessary.

System V has kernel support routines available for device drivers that handle 286 system calls. For example, **ldtalloc**(K), **ldtfree**(K), **cvtoint**(K), and **cvtoaddr**(K) are provided as stubs to help facilitate compilation. These stubs may be removed in any future release of System V. Refer to the "Obsolete Kernel Routines" section in this appendix for information on other stubbed or dropped kernel routines.

Change in the Device Number

In System V, the device number now contains both the major and the minor number and is a short integer of type **dev_t**. Use the **major**(K) kernel routine to extract the major device number; use the **minor** kernel routine to extract the minor device number. These two routines are described on the **major**(K) manual page in the section (K) manual pages in this guide.

Changes in Structure Definitions

Structure Header Files

In System V, driver-related header files are in */usr/include/sys*. Use of the old path for header files is supported, but all new drivers should be written to take advantage of the new path. Header files included in a driver with the following syntax access header files in the new path:

#include "*sys/filename.h*"
#include <*sys/filename.h*>

Changes to the buf Structure

The following changes were made to the *buf* structure in System V:

Field	**Description**
b_blkno	This field is now in units of 512 bytes instead of 1024 bytes. This field is read only (cannot be set by a driver).
b_cylin	This field is no longer being used by **disksort**(K) for sorting requests. The preferred field is the new *b_sector* field because *b_sector* is a longer (signed long) field. (*b_cylin* is a ushort and is read only.) CAUTION: Assigning *b_sector* to *b_cylin* can cause sign extension. Use the following call when converting: `bp->b_sector = (daddr_t)bp->b_cylin & 0xFFFF`
b_flags	New flag values have been added and two flags have been dropped, but neither the added or dropped flags have any bearing on drivers. The added flags are: B_VERIFY, B_FORMAT, B_REMOTE, B_S52K, and B_PRIVLG. The dropped flags are B_NOCROSS and B_FLUSH.

Refer to *sys/buf.h* for more information. The *b_flags* field is driver setable, but must never be cleared.

b_paddr

This field has been replaced in the *buf* structure by the *b_un* union that has these read-only fields:

```
caddr_t b_addr;     /* low order core address */
int *b_words;       /* words for clearing */
daddr_t *b_daddr    /* disk blocks */
```

The *b_paddr* field under XENIX stored a physical address. The *b_un.b_addr* field stores a virtual address. Use **paddr**(K) to access this field. Note that **paddr** now returns a virtual address. The following call returns a physical address:

```
paddr = ktop(paddr(bp));
```

as does:

```
vtop(paddr(bp), b_proc);
```

Replace occurrences of *b_paddr* with calls to **ktop**(K) or **vtop**(K) where appropriate.

b_sector

This field is new in System V and is the physical sector of the disk request. This field is typed as *daddr_t* (long) and is read only. Be careful of sign extension when replacing *b_cylin* references with *b_sector*. The following call prevents sign extension:

```
bp->b_sector = (daddr_t) bp->b_cylin & 0xFFFF;
```

b_start

This field is new in System V and is the request start time. This field is typed as **time_t**, and can be set by a driver.

b_proc	This field is new in System V and is a pointer to a user processes *proc* structure. This field is an instance of struct *proc* and is read only.
b_reltime	This field is new in System V and is the previous release time. This field is typed as an unsigned long and is read only.
b_want	This field is new in System V and stores the B_WANTED flag. The comment for this field states: "Need this field to hold the B_WANTED flag in order to avoid a race in the updating of *b_flags*. A process with a buffer locked assumes it can do whatever it likes with *b_flags*, while a process wanting a busy buffer sets the B_WANTED bit without having the buffer locked. Moving it to a separate word avoids problems with simultaneous updates."

In addition, the *buf* structure has been *typedef*-ed as *struct buf_t*.

user Structure

Under System V, the *user* structure no longer supports the *u_cpu* field. A new field in the *user* structure, *u_renv*, contains the same information as *u_cpu* in bits 16-23.

The XENIX *u_baseu* field is now called *u_base*.

cblock Structure

In System V, there is no longer a *d_flags* field in the *cblock* structure. This affects line discipline drivers that use delays.

devbuf Structure

The *devbuf* structure (as defined in */usr/include/sys/devbuf.h*) has changed as follows:

- *bufptr* and *bufend* are now offsets (because the kernel has been relocated into a higher address space).

  In XENIX, if *bufptr* = 4, then the block actually started at 4*512 = 2K. In System V, *bufptr* and *bufend* (as well as *head* and *tail*) are now physical addresses. If *bufptr* =1K, it is physically 1K bytes into RAM starting from RAM address 0.

- In System V, only one *devbuf* structure can be allocated with **db_alloc**(K), in XENIX up to 30 *devbuf* structures could be allocated in a single call to **db_alloc**.

  Example

```
struct devbuf dbuf[2];      /* XENIX */
                .
                .
        db_alloc(&dbuf,2);

struct devbuf dbuf, *dptr;  /* System V */
                .
                .
        dptr = &dbuf;
        db_alloc(dptr);
```

Changes in Kernel Routines

physio(K)

The **physio** routine has changed. XENIX checked to see if a buffer passed to **physio** was busy. If it was, **physio** allocated a private buffer for the driver. In System V, you must make sure that the buffer is not busy (B_BUSY is set in *b_flags*), before calling **physio**. If a busy buffer is given to **physio**, a catastrophe results. To get **physio**-owned buffers, use the following call format:

```
physio(xxbreakup, (struct buf *) NULL, dev, rwflag);
```

Setting the second argument to NULL instructs **physio** to allocate its own buffers.

In System V, the **xxread** and **xxwrite** routines should first call the **physck**(K) routine to validate the requested transfer. The syntax is:

int physck(nblocks, rw);
daddr_t nblocks;

In this example, *nblocks* is the number of sectors that the device has. The routine knows the size of the user's request because it examines the fields *u.u_offset* and *u.u_count*. So only the total size of the device needs to be supplied.

Next, call **physio** with a pointer to the device driver's **xxbreakup** routine. **xxbreakup** should then call the system breakup routine with a pointer back to the **xxstrategy** routine. The system breakup routine is one of either **dma_breakup**(K), **pio_breakup**(K), or a user supplied breakup routine. The **pio_breakup** routine should generally be used to break a request across page boundaries.

The following floppy driver example illustrates the breakup conventions under System V:

```
flread(dev)
dev_t dev;
{
register int size;

size = flblktosec(flsize[sizeindx(dev)]);    /* size in sectors */
if (physck(size, B_READ))
physio(flbreakup, NULL, dev, B_READ);
}

flbreakup(bp)
struct buf *bp;
{
int flstrategy();

dma_breakup(flstrategy, bp);
}
```

disksort(K)

Under System V, the **disksort** routine uses the *b_sector* field of the *buf* structure to sort requests. The *b_sector* field is a 32 bit field, which gives better resolution than the 16 bit *b_cylin* field. Under XENIX, **disksort** uses the *b_cylin* field.

xxopen, xxclose, xxread, xxwrite, xxioctl

Under System V, the **xxopen**, **xxclose**, **xxread**, **xxwrite**, and **xxioctl** routines are called with the entire device number. XENIX calls these routines with the minor device number only.

major(K), minor(K) Device Number Macros

When converting drivers, you can extract either the major or the minor device numbers by using the **major** and **minor** macros to mask out the unneeded portions. The **major** macro returns the major device number, while the **minor** macro returns the minor number. To ascertain the entire device number, use the instruction:

```
minornum = minor(dev);
majornum = major(dev);
dev = makedev(majornum, minornum,);
```

cmn_err(K)

The first argument to **cmn_err** has been dropped for 3.2. Also there is no **CE_INFO** arg to System V **cmn_err** as there was for XENIX **cmn_err**. For example:

```
XENIX:

cmn_err("serial", CE_WARN, "interrupt vector collision");

System V:

cmn_err(CE_WARN, "serial: interrupt vector collision");
```

Note

cmn_err causes a panic if the wrong version is used.

---

sptalloc(K) and sptfree(K)

Both **sptalloc** and **sptfree** have changed. For example:

```
XENIX:  char *sptalloc(size)
       int size;
       /* size is in bytes */

       sptfree(ptr, size, flag)
       char *ptr;
       int size, flag;
       /* to actually free pages, flag always = 1 */


System V:     sptalloc(size, mode, base, flag)
       int size, mode, base, flag;
       /* size is in clicks (pages) */
       /* mode is defined in immu.h */
       /* for base and flag, see sptalloc(K) manual page */

       sptfree(vaddr, size, flag)
       char *vaddr;
       int size, flag;
       /* size is in pages */
       /* flag is always 1 to free pages */
```

In System V, **sptalloc**(K) is used for mapping device registers to I/O mapped memory. For example, to map the device registers of a card to address 0xB8000, use this **sptalloc** call:

```
   vaddr = sptalloc(1, PG_P, 0xB8000, 1);
```

Obsolete Kernel Routines

The following kernel routines are no longer necessary nor are they documented in the manual pages section in this guide. These routines, unless indicated differently, are available in the kernel, but only return errors when called and are only provided for compatibility purposes. The routines are:

Routine	*Notes*
cvttoaddr	Convert far address to 286 virtual address. Reason for being deleted: not applicable on 386. It is no longer necessary to use **cvttoaddr** to convert an 8086/80286 address to a valid 386 kernel virtual address because every binary is considered to be a 386 binary by the System V kernel. An emulator determines if the binary in question is actually a 286 or 86 binary. Although a stub for **cvttoaddr** has been provided, it may be removed in a later release.
cvttoint	Extract low 16 bits from a 286 far pointer. Reason for being deleted: not applicable on 386.
dscraddr	Get physical address. Replaced by **sptalloc**(K).
dscralloc	Allocate GDT descriptor. Reason for being deleted: not applicable on a 386.
dscrfree	Free GDT descriptor. Reason for being deleted: not applicable on a 386.
ftoof	Convert a number from an address to an offset. This routine is not available in the System V kernel in any form. Reason for being deleted: not needed on a 386.
ftoseg	Convert a number from an address to a segment. This routine is not available in the System V kernel in any form. Reason for being deleted: not needed on a 386.
getcbc	Return the first character of a cblock. This routine is not available in the System V kernel in any form. The functionality of this routine is handled by the **getc**(K) routine.

in	Read a 16-bit word from a physical address. This routine is not available in the System V kernel in any form. Replaced by **inb**(K).
IS386	Determine if a process is being run on a 386 computer. No replacement. System V only runs on a 386 computer. This routine is still usable, but is unnecessary.
ldtalloc	Allocate free user mapping descriptors. Reason for being deleted: not applicable on a 386.
ldtfree	Release allocated descriptor. Reason for being deleted: not applicable on a 386.
mapphys	Map physical address to a kernel virtual address. Replaced by the **vas**(K) routines and **sptalloc**(K).
mapptov	Map physical addresses to specific virtual addresses. Replaced by the **vas**(K) routines.
out	Write a 16-bit word from a physical address. This routine is not available in the System V kernel in any form. Replaced by **outb**(K).
putcbc	Copy characters to a *clist*. This routine is not available in System V in any form. The functionality of this routine is handled by the **putc**(K) routine.
setjmp	The **setjmp**(K) routine is no longer documented. Use of this routine can cause system corruption or panic. This routine is not similar in any way to the **setjmp**(S) system call.
sotofar	Converts segment and offset numbers. This routine is not available in System V in any form. Reason for being deleted: not necessary on a 386.
unmapphys	Unmap an address previously mapped with **mapphys**. Replaced by the **vas**(K) routines and **sptfree**(K).

General Notes

Halt Routines and Interrupts

Any driver that controls hardware should check for pending interrupts on its channel before running any **xxhalt** routine defined in **io_halt**[ ]. When there are no interrupts pending, interrupts should be suspended. Then, the **xxhalt** routine may be executed and interrupts may be enabled again.

Include Files

All XENIX include lines of the form:

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

must be changed to the form:

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

          or

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

cram.h

There is a new include file, *sys/cram.h* that contains **#define** statements dealing with programming the CMOS.

Near and Far Keywords

All references to the **near** and **far** keywords must be removed for System V. In addition **faddr_t** references should be changed to **caddr_t**, even though the latter is provided for compatibility purposes in *sys/types.h*.

xxprint Driver Routine

For future compatibility, all block device drivers should have an **xxprint** routine. The following example shows a sample **xxprint** routine for a floppy diskette driver:

```
xxprint(dev, str)
dev_t dev;
char *str;
{
   cmn_err (CE_NOTE, "%s on floppy diskette unit= %d, minor= %d",
            str, unitbits(dev), minor(dev));
}
```

In the previous example, **unitbits** is a macro used only by the floppy disk driver for determining which device is selected. The macro is defined as follows:

```
#define unitbits(dev)     ((dev) & 0x3)
```

machdep.h

There is no *machdep.h* file in System V. This file has been split up into the driver-specific *.h* files in the */usr/include/sys* directory.

Compiler Directives

The compiler directives **-Di386** and **-Dunix** are enabled in this release. Using the **-Dunix** compiler directive is required in order to compile for System V.

#ifdef M_S_UNIX

Use the statement **#ifdef M_S_UNIX** in System V drivers that will be used in both XENIX and System V. Place the **#ifdef** statements around System V specific code.

disk.c

disk.c has changed as follows:

- *b_blkno* is now in units of 512 bytes instead of 1K bytes.
- **maxvec** is now equal to the integer variable *nintr*, in XENIX it was equal to the constant "MAXVEC."
- **nswap** under System V is in 512 byte blocks, in XENIX it was in 1K byte blocks.

Interrupt Vectors

The numbering of interrupt requests (IRQ's) on the slave chip has changed. All numbers shown are in decimal. The values are:

XENIX:	master 0-7	System V:	master 0-7
	slave 24-31		slave 8-15

The above table is in reference to the indices into **vecintsw** and **ivect** in XENIX and System V respectively. Note that **vecintsw** does not exist in System V.

Possible Problem Areas

- Sign extension problems may occur, due to the kernel being moved to a much higher address space. This is because the kernel has the high bit of all its virtual addresses set to "on."
- Physical versus virtual addresses: in System V, **ptok** and **ktop** must be used to convert from physical to kernel virtual addresses, and from kernel virtual to physical addresses. Under XENIX, these macros had no effect and were sometimes omitted; this is not the case under System V.

A

Converting IDDs

Overview

When converting a XENIX installable device driver (IDD) to System V, note that the organization of the System V file system structure differs from XENIX. Some XENIX files have been logically split up into functionally separate files. For example, the file *master* has been split into *mdevice* and *mtune* to separate device information from tunable variable information. There is also a much more liberal use of directories to keep functionally different files separate. There are several key differences where there is a mapping of functionality from XENIX to System V.

XENIX Files Relocated Under System V

Below are the major functional groups of files. Under each group there are examples of XENIX files and their functionality in relation to their System V counterparts.

XENIX	System V
Reconfiguration root directory:	
/usr/sys/	*/etc/conf/*
Executable files:	
conf/configure	*cf.d/configure*
conf/link_xenix	*cf.d/link_unix*
Configuration files:	
conf/master	*cf.d/mdevice*
	cf.d/mtune
conf/xenixconf	*cf.d/sdevice*
	cf.d/stune
Driver specific files:	
io/sioconf.o	*pack.d/sio/Driver.o*
io/sioconf.h	*pack.d/sio/sioconf.h*

Summary of XENIX vs. System V Directory Structures

Below are the relevant directory structures of both XENIX and System V. On the left is the directory name and on the right a brief explanation of its contents.

XENIX

/usr/sys/	Root reconfiguration directory.
/usr/sys/conf/	Configuration dependent files and installable device commands.
/usr/sys/h/	Include files used in recompilation.
/usr/sys/io/	I/O driver object files, libraries and structure definitions.
/usr/sys/mdep/	Machine dependent library.
/usr/sys/sys/	System library.

System V

/etc/conf/	Root reconfiguration directory.
/etc/conf/bin/	Stores ATT style installable device commands.
/etc/conf/cf.d/	Configuration dependent files.
/etc/conf/pack.d/	Contains one directory per Driver Software Package (DSP).
/etc/conf/pack.d/{DSP_prefix}/	Driver object files and structure definitions.
/etc/conf/rc.d/	Startup for each DSP. Linked to *etc/idrc.d.*
/etc/conf/sd.d/	Shutdown for each DSP. Linked to *etc/idsd.d.*
/etc/conf/sdevice.d/	Contains one file with all configuration entries for each type of device.
/etc/conf/node.d/	ATT style device node definitions for each DSP.
/etc/conf/init.d/	Contains "/etc/inittab" entries for each DSP.
/etc/conf/mfsys.d/	One FS master data file per file system type add-on.
/etc/conf/sfsys.d/	One FS system data file per file system type add-on.
/usr/include/sys/	Include files used in recompilation.

Converting Installation Scripts

In addition to the new directory structure, the System V approach stresses modularity. To simplify conversion, these aspects of the installation are hidden by the extended functionality of the new **configure**(ADM) command.

For converting an installation, here is a list of appropriate corresponding XENIX and System V commands followed by an optional description or explanation of each command:

Extracting files:

XENIX: **custom**
System V: **custom**

Follow the menu to extract the files in the package.

Check to see if the device is already installed:

XENIX: **configure -j** {*device prefix*}
System V: **configure -j** {*device prefix*}

Returns 1 if no conflicting device name is found.

Adjust tunable kernel parameters:

XENIX: **configure** {*resource*}={*value*}
System V: **configure** {*resource*}={*value*}

Resource is the name of the tunable parameter.

Check for address or interrupt vector conflicts:

XENIX: **vectorsinuse**
System V: **configure -V** {*vector*}
System V: **configure -A** {*low_address, high_address*}

vectorinsuse prints out a list of vector numbers currently in use. To check for address conflicts, **awk**(C) is generally used to interrogate the master file. **configure -V** returns non-zero if there is a conflict. **configure -A** returns non-zero if there is an address conflict within the given range.

Find the next available major number:

XENIX: **configure -j** *NEXTMAJOR*
System V: **configure -j** *NEXTMAJOR*

Change configuration files:

XENIX: **configure -b -c -m** {*maj_dev_num*}
-v {*interrupt vector number*}
-a {*list of routines*}
-l {*interrupt priority level*}

System V: **configure -b -c -m** {*maj_dev_num*}
-v {*interrupt vector number*}
-a {*list of routines*}
-l {*interrupt priority level*}
-h {*device prefix*}
-Y

Note that **-b** and **-c** denote block and/or character devices.

The differences to be noted are the **-h** option followed by the device prefix and **-Y** to include this configuration into the new kernel under System V.

Edit link_xenix:

XENIX: **sed** or **vi**
System V: The *link_unix* script is not edited.

Under XENIX you add the names of all the new object files to the **ld** command line in this file. This is not necessary under System V.

Relink the kernel:

XENIX: **link_xenix**
System V: **link_unix**

Create a device node:

XENIX: **mknod** {*name*} {*(b)lock or (c)haracter*} {*major number*} {*minor number*}
System V: **mknod** {*name*} {*(b)lock or (c)haracter*} {*major number*} {*minor number*}

Install kernel in default location:

XENIX: **hdinstall**
System V: No corresponding command.

Under System V, **hdinstall** is accomplished through **link_unix**.

Invoke the new kernel:

XENIX:**shutdown**
System V: **shutdown**

Appendix B

Sample Block Driver

Overview B-1

Block Driver Header File B-2

Block Driver B-4

Overview

This appendix contains a sample block driver for a 96 tracks per inch (tpi) double-sided high-density (dshd) floppy disk drive. This driver provides read and write acces to raw and block data, but does not provide any formatting capabilities.

Use the index at the end of this book to locate structures and routines provided in the example driver. The following table lists the line numbers on which each driver routine appears:

Routine	*Line*	*Description*
blckopen(dev, mode, flag)	187	Open device
blckclose(dev, mode, flag)	202	Close device
blckstrategy(bp)	240	Access buffer header
blckprint(dev, str)	1000	Print error message
blckinit( )	173	Initialize device
blckstart( )	275	Interact with device
blckhalt( )	974	Called on shutdown
blckintr( )	356	Interrupt handler
blckbreakup( )	730	Break up I/O request
blckioctl(dev, cmd, uargp, flag)	218	IO control commands
blckread(dev)	735	Raw I/O read
blckwrite(dev)	741	Raw I/O write
blck_dma( )	548	DMA d_proc routine

Block Driver Header File

```
/*
 *
 *       Copyright (C) 1989 The Santa Cruz Operation, Inc.
 *
 */

#define FL_ERROR  -1
#define WP_ERROR  -2
#define WR_PROT   0x40  /* write protect bit of drive stat*/

 /* Floppy Disk Controller port addresses */

#define P_FLCTL   0x03f2   /* FDC control register (write)*/
#define P_FLMSR   0x03F4   /* FDC main status  register   */
#define P_FLDCR   0x03F5   /* FDC data control register   */
#define P_FLDCT   0x03F7   /* FDC control register (write)*/
#define P_FLDMA   0x0112   /* DMA data port */
#define P_FLTC    0x0140   /* DMA terminal count port */

#define FDC_ENB   0x04     /* high(FDC enable) low(reset) */
#define FL_ENB_DI 0x08  /* enable floppy DMA & interrupts */

        /* MSR bits for Floppy */

#define DA_BUSY 0x01  /* Drive A seeking */
#define DB_BUSY 0x02  /* Drive B seeking */
#define DC_BUSY 0x04  /* Drive C seeking */
#define DD_BUSY 0x08  /* Drive D seeking */
#define CB      0x10  /* FDC busy */
#define NON_DMA 0x20  /* Non DMA mode */
#define DIO     0x40  /* Data I/O: low(write) high(read) */
#define RQM     0x80  /* Request for master - polled  */
                      /* before writing control bytes */
#define SDRV_RDY 0x20 /* Drive ready bit of status byte 3 */
#define SWR_PROT 0x40 /* Write protected bit */
                      /* of status byte 3 */
        /* Floppy command codes */
#define RD_TRK      (0x02) /* Read a track (floppy) */
#define SPECIFY     (0x03) /* Specify */
#define SENS_DR     (0x04) /* Sense drive status */
#define WR_DATA     (0x05) /* Write data */
#define RD_DATA     (0x06) /* Read data */
#define RECAL       (0x07) /* Recalibrate */
#define SENS_INTR   (0x08) /* Sense interrupt status */
#define WR_DEL_DATA (0x09) /* Write deleted data */
#define RD_ID       (0x0A) /* Read ID */
#define RD_DEL_DATA (0x0C) /* Read deleted data */
#define SEEK        (0x0F) /* Seek */
#define FORMAT      (0x0D)
#define MT           0x80  /* multi-track */
#define MF           0x40  /* FM mode=0,MFM=1(dbl density)*/
#define SK           0x20  /* skip deleted data mark */

/* Table for number of control writes and status reads */

#define NCMDS   16 /* Number of commands */
#define MAX_WR   9 /* Maximum number of control writes */
#define MAX_RD   7 /* Maximum number of control reads */

/* Next defines are indexes into cmd array elements below */

#define NUM_WR   0 /* Number of control writes element */
#define NUM_RD   1 /* Number of status reads element */
#define EXEC     2 /* If cmd has execution phase */
                   /* with resulting interrupt    */

/* Floppy table */

int fl_cmd[NCMDS][3] = {

/* Code             Write     Read     Execute */
/***************************************************/
/* Invalid     */    {0,       0,         0},
/* Invalid     */    {0,       0,         0},
/* RD_TRK      */    {9,       7,         1},
/* SPECIFY     */    {3,       0,         0},
/* SENS_DR     */    {2,       1,         0},
/* WR_DATA     */    {9,       7,         1},
/* RD_DATA     */    {9,       7,         1},
/* RECAL       */    {2,       0,         1},
/* SENS_INTR   */    {1,       2,         0},
/* WR_DEL_DATA */    {9,       7,         1},
/* RD_ID       */    {2,       7,         1},
/* Invalid     */    {0,       0,         0},
/* RD_DEL_DATA */    {9,       7,         1},
/* FORMAT      */    {6,       7,         1},
/* Invalid     */    {0,       0,         0},
/* SEEK        */    {3,       0,         1}
};
```

Block Driver

Use the index at the end of this book to locate routines and structure names in this driver.

```
/*
 *
 *   Copyright (C) 1989 The Santa Cruz Operation, Inc.
 *
 *
 *   This driver is designed for the NEC 765A
 *   (Intel 8272A) controller.  96 tpi, and 15 sectors
 *   per track is supported.
 */

#define  near                   /* no 'near' keyword */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/systm.h"
#include "sys/buf.h"
#include "sys/iobuf.h"

#include "sys/cram.h"

#include "sys/dir.h"
#include "sys/file.h"
#include "sys/signal.h"
#include "sys/seg.h"
#include "sys/page.h"
#include "sys/immu.h"        /* must come before region.h */
#include "sys/region.h"
#include "sys/proc.h"
#include "sys/x.out.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/open.h"
#include "sys/floppy.h"
#include "sys/conf.h"
#include "sys/cmn_err.h"
#include "sys/dma.h"
#include "sys/vendor.h"
#include "blck.h"
```

```
#define CE_ERROR    CE_WARN
#define MAXRETRY    12
#define CNTRL_WR(x) outb(P_FLCTL,x) /* write control port */
#define FL_CHANGE    2
#define TRUE        -1
#define FALSE        0

/*
 * a bit set in recalstat means that the drive
 * has been recalibrated since the last reset;
 * bit clear means recalibration must be done.
 */

int     recalstat = 0 ;

struct  dmareq dma_q;           /* DMA request structure */

/* current state in floppy operation - should          */
/* be an blcktab variable?                             */
int        flstate = -1 ;

/* defines for the above state variable */
#define STRESET     0
#define STRECAL     1
#define STSEEK      2
#define STIO        3
#define STFMSEEK    4
#define STMOTOR     6

int     flinitflg = TRUE,  /* hardware setup must be done */
        flerreset = FALSE; /* must do reset for error     */
                           /* recovery */

long     fl_ma;            /* Machine address (long       */
                           /* physical address)           */
unsigned fl_lsn,           /* logical sector number       */
         fl_tn,            /* track number                */
         fl_sn,            /* sector number               */
         fl_eot;           /* last sector number          */

int      fl_to,            /* motor timeout flags         */
         fl_job;

unsigned fltranscnt,   /* number of bytes in this transfer*/
         flresidcnt,   /* number of bytes remaining to    */
                       /* transfer                        */
         fltrnsferred, /* total number of bytes           */
                       /* already transferred             */
dor = 0;               /* memory image of DOR             */

#define FLSECSZ  512   /* sector size in bytes            */
#define FLSMASK  511
#define FLSHIFT    9
```

```
/*
 * If a transfer crosses a 64k boundary in memory, DMA it
 * in/out of kernel data and copy it to/from user memory.
 */

int      fllocaltransfer;       /* flag this transfer as     */
                                /* a local one               */
char     fllocalbuf[FLSECSZ];  /* buffer used for transfer */
paddr_t fltmpaddr;     /* hold paddr during local transfer */

/* convert bytes to sectors */
#define flbtos(x)   (((x)+FLSECSZ-1)>>FLSHIFT)

/* convert sectors to bytes */
#define flstob(x)   ((x)<<FLSHIFT)

char     fl_cmds[MAX_WR];  /* Buffer for blckcommand bytes */
char     fl_stats[MAX_RD]; /* Buffer for status reads      */

struct  iobuf   blcktab;  /* iobuf defined in sys/iobuf.h */
struct  buf     rblckbuf; /* buf is defined in sys/buf.h  */

/* minor device encoding */
#define SIZEBITS         0x3C
#define TPI96BIT         0x20    /* 1=>80 tracks          */
#define SECBITS          0x18    /* 10->15 sectors/track  */
#define DSBIT            0x04    /* 1=>double sided       */
#define UNITBITS         0x03    /* physical drive        */
#define unitbits(dev)    ( (dev)        & 0x3)
#define sizebits(dev)    (((dev) >> 2)  & 0xF)
#define secbits(dev)     (((dev) >> 3)  & 0x3)
#define fl_onecyl(dev)   ((dev & DSBIT ? 2 : 1) * 15)

unsigned flhead;                /* head number, 0 or 1       */

/* defines for block/char specification to blckopen         */
#define CHRDEV  OTYP_CHR        /* 3rd parameter to device  */
#define BLKDEV  OTYP_BLK        /* open routine specifying  */
                                /* block or char device     */

unsigned int    flcopened = 0,           /* count of opens */
                flbopened = 0;           /* count of opens */

/*
 * format buffer is to be used for a collection of desired
 * address fields for the track.  Each field is composed of
 * 4 bytes, (C,H,R,N), where C = Track number,
 * H = Head number, R = Sector number,
 * N = number of bytes per sector ( 02=512 )
 * There must be one entry for every sector on the track.
 * This information is used to find the requested sector
 * during read/write access.
 */
```

```
struct sctrhdr {
        char    sc_trk;     /* track number                  */
        char    sc_hd;      /* head number (0 or 1)          */
        char    sc_sec;     /* sector number (1-18)          */
        char    sc_nb;      /* numbytes/sector               */
} fl_fmtbuf[18];            /* format byte field buffer      */

/* defines and structs for ioctl calls                       */
#define FLBUSY     1        /* device busy                   */
#define FLNOWR     2        /* device not opened for writing */
#define FLILLTN    3        /* illegal track number          */
#define FLNOTDS    4        /* not double sided,             */
                            /* illegal side referenced       */
#define FLIOERR    5        /* device error during           */
                            /* requested operation           */

struct fmtfl {              /* ioctl struct format floppy    */
    char    fm_trk;         /* track number                  */
    char    fm_hd;          /* side number                   */
    char    fm_sec;         /* sector number (future use)    */
    char    fm_size;        /* code: bytes/sector-future use */
    char    fm_il;          /* interleave                    */
    char    fm_status;      /* return status, 0 good         */
};

/* controller/floppy parameters */
struct  flptab {
    char    fl_spc1;        /* 1st spec byte                 */
    char    fl_spc2;        /* 2nd spec byte                 */
    char    fl_mont;        /* motor-on  delay time (16 ms)  */
    char    fl_moft;        /* motor-off delay time (sec)    */
    char    fl_nbs;         /* bytes/sector code             */
    char    fl_dtl;         /* special sector length code    */
    char    fl_gpl;         /* gap length                    */
    char    fl_gplf;        /* gap length for format         */
    char    fl_filf;        /* filler byte for format        */
};
```

```
/*
 *  Default floppy hardware parameters.
 */

struct  flptab        flparam = {
    /* spc1 */        (0x0D<<4)|(0x0F),
    /* spc2 */        (0x08<<1)|(0x00),          /* 16ms HLT  */
    /* mont */        8,          /* 8 * 1/8 sec = 1 second */
    /* moft */        3,                         /* 3 seconds */
    /* nbs  */        2,                         /* 512 bytes */
    /* dtl  */        0xFF,
    /* gpl  */        0x1B,
    /* gplf */        0x54,
    /* filf */        0xF6
};

static char fl_name[] = "DMA blck";

int blckintr();
blckinit()
{
        int  base, offset, type;
        extern int        (*ivect[])();
        extern int        nintr;

        base = 0x03f2;
        offset = 0x05;

        printcfg(fl_name, base, offset, 6, 2, "unit=0 %s",
                                "sample blck 96tpi");
}

/*
 * blckopen() - Increment open count and check
 *              for various errors.
 */

blckopen(dev, mode, flag)
{
            switch(flag)
            {
                    case CHRDEV:
                            flcopened++;
                            break;
                    case BLKDEV:
                            flbopened++;
                            break;
            }
}
/*
 * blckclose() - Set open count to 0 for device
 */
```

```
blckclose(dev, mode, flag)
{
               switch(flag)
                         {
                         case CHRDEV:
                                   flcopened = 0;
                                   break;
                         case BLKDEV:
                                   flbopened = 0;
                                   break;
                         }
}

/*
 * blckioctl()
 *
 */

blckioctl(dev, cmd, uargp, flag)
caddr_t    uargp;
{
          struct fmtfl         sfmtfl;

          switch(cmd)
          {
          case FLIOCSIZE:
                    sfmtfl.fm_trk    = 80;
                    sfmtfl.fm_hd     = 2;
                    sfmtfl.fm_sec    = 15;
                    sfmtfl.fm_size   = flparam.fl_nbs;
                    sfmtfl.fm_il     = 1;
                    sfmtfl.fm_status = 0;
                    if ( copyout(&sfmtfl,uargp,
                              sizeof sfmtfl) == -1 )
                              u.u_error = EFAULT;
                    break;
          default:
                    u.u_error = EINVAL;
                    break;
          }
}
```

```
blckstrategy(bp)
register struct buf *bp;
{
            /* Check valid block */
            if((bp->b_blkno < 0)||(bp->b_blkno > 2400))
            {
                    bp->b_flags |= B_ERROR ;
                    bp->b_error = ENXIO ;
                    bp->b_resid = bp->b_bcount ;
                    iodone(bp);
                    return;
            }

            /* EOF case - starting block is just at end */
            if (bp->b_blkno == 2400)
            {        bp->b_resid = bp->b_bcount ;
                    if ((bp->b_flags & B_READ) == 0)
                    {
                            bp->b_flags |= B_ERROR ;
                            bp->b_error = ENXIO ;
                    }
                    iodone(bp);
                    return ;
            }
            devque( bp );
}
```

```
/*
 * blckstart() -  Part of the state machine is here and
 * part is in blckintr().  blckstart() needs to set
 * variables for multi-block transfers.  Note that
 * the "first time reset" is here based on initflg)
 * rather than start or open; this is because the reset
 * generates an interrupt and it is preferable to have
 * the I/O variables for the transfer already set up.
 * Motor is turned on here if necessary.
 */

blckstart()
{
        register struct buf *bp ;
        register overflow;      /* number of sectors a    */
                                /* raw transfer overflows */
        extern blck_timeout();

        /*
         * if "previous" request was a floppy, make sure
         * the timeout chain is running; this covers the
         * case of a floppy timeout error where a delay
         * is still desired prior to turning off motors....
         */
        if( fl_to == 0 )
        {
           timeout( blck_timeout, 0, Hz );
           fl_to = 1;
        }

        if ( (bp = blcktab.b_actf) == NULL)
        {
           dma_relse ( DMA_CH2 );
           blcktab.b_active = 0;
           return;
        }

        blcktab.b_active = 1;
        fl_lsn = bp->b_blkno;

        /* use vtop instead of ktop; paddr(bp) can be a   */
        /* user address such as when called by physio     */

        fl_ma  = vtop(paddr(bp), bp->b_proc);

        blckmap() ;        /* set flhead, fl_sn, fl_tn    */
        /* Bad starting block and EOF were checked in
         * strategy.  The ability to do only a partial
         * transfer of a record (raw) is checked here.
         */

        overflow = fl_lsn + flbtos(bp->b_bcount) - 2400;
        flresidcnt = bp->b_bcount ;
        if (overflow > 0)
                flresidcnt -= flstob(overflow);

        fltranscnt = 0 ;
        fltrnsferred = 0 ;

        if (flinitflg)
        {
           /* initial reset hasn't been done */
            flinitflg = 0 ;
            flstate = STRESET ;
            blckreset() ;
            return;  /* enter state machine on interrupt */
        }

        if (flerreset)
        {
            flerreset = 0;  /* last transfer had an error */
            flstate = STRESET ;
            blckreset() ;
            return ;  /* enter state machine on interrupt */
        }

        if (motoron() == 0)
        {
                timeout(blckstart, 0,
                        (flparam.fl_mont * Hz) / 8);
                return;
        }

        /* check if a recalibrate is necessary */
        if (recalstat == 0)
        {
                flstate = STRECAL ;
                blckcommand(RECAL);
                return;
        }

        flstate = STSEEK ;
        blckcommand(SEEK);
        return ;
}
```

```
/* blckintr(): if returning from a recalibrate, seek, or
 * reset command, issue a sense interrupt status command
 * to check for errors.  On commands such as read and
 * write that have a result phase, use cmdresult( ).
 */

blckintr()
{
        register struct buf *bp;
        register int stat ;

        if( (bp = blcktab.b_actf) == NULL )
        {
                if( fl_to == 0 )
                {
                    extern int blck_timeout() ;
                    timeout( blck_timeout, 0, Hz );
                    fl_to++ ;
                }
                if (flstate == STRESET)
                    blckcommand(SPECIFY);
                return ;
        }
        if (flstate == STIO)
        {
                if ( cmdresult( (bp->b_flags & B_READ)
                                  ? RD_DATA : WR_DATA)
                             == FL_ERROR )
                        goto error ;
                if ( cmd_status() == FL_ERROR )
                {
                        if ((fl_stats[1]&0xFF) != 0x80)
                                goto error;
                }
                /*
                 * if transfer is local, local routine
                 * picks it up and kicks it off again.
                 */
                if (fllocaltransfer) {
                        blcklocal(bp);
                        return;
                }
                flresidcnt -= fltranscnt ;
                fltrnsferred += fltranscnt ;
                if (flresidcnt == 0)    /* no more left */
                        goto done ;

                /* continue transferring */
                fl_ma  += fltranscnt ;
                fl_lsn += flbtos(fltranscnt) ;

                blckmap() ;   /* set fl_sn, fl_tn, flhead */
                goto seek ;
        }
```

(*Continued on next page.*)

```
        /* if floppy driver controller caused interrupt */
        if (flstate != STMOTOR)
        {
                blckcommand(SENS_INTR);

                /* ready change on a reset command is ok */
                stat = cmd_status();

                /* stat return on RESET must be CHANGE */
                /* any other non-zero stat is an error */
                if (flstate == STRESET)
                {
                    if (stat != FL_CHANGE)
                        goto error ;
                }
                else
                {
                    if (stat)
                        goto error ;
                }
        }

        switch(flstate)
        {
                case        STRESET:
                                blckcommand(SPECIFY);
                                if(motoron() == 0)
                                {
                                    timeout(blckintr, 0,
                                        (flparam.fl_mont
                                                * Hz) / 8);
                                    flstate = STMOTOR;
                                    return;
                                }
                                /* fall through ... */
                case        STMOTOR:
                                flstate = STRECAL ;
                                if (blckcommand(RECAL) ==
                                                FL_ERROR)
                                        goto error;
                                return ;
```

(*Continued on next page.*)

```
                case        STRECAL:
seek :
                                flstate = STSEEK ;
                                if (blckcommand(SEEK) ==
                                                FL_ERROR)
                                        goto error;
                                return;

                case        STSEEK  :
                                flstate = STIO;
                                stat = blckxfer(bp);
                                if (stat == WP_ERROR)
                                {
                                    bp->b_flags |= B_ERROR ;
                                    flerreset = TRUE ;
                                    goto done;
                                }
                                if (stat == FL_ERROR)
                                {
                                    goto error;
                                }
                                return ;
                default:
                                goto error;
        }
```

(*Continued on next page*.)

```
error :

        if (fllocaltransfer) {
                fl_ma = fltmpaddr;
                fllocaltransfer = 0;
        }

        if (++blcktab.b_errcnt < MAXRETRY)
        {
                flstate = STRESET;
                blckreset();
                return ;
        }
        /* hard error */
        bp->b_flags |= B_ERROR ;
        /* have to do reset before next transfer */
        flerreset = TRUE ;

done  :
        blcktab.b_actf = bp->av_forw ;
        blcktab.b_errcnt = 0 ;
        blcktab.b_active = 0 ;
        bp->b_resid = bp->b_bcount - fltrnsferred ;
        iodone(bp) ;
        blckstart() ;
}

/***************************************************
 * Driver Utilities
 *
 */


/*
 * blckmap() - routine to set the globals fl_tn,
 * fl_sn, and flhead from the global fl_lsn.
 * Called from start and intr.
 */
blckmap()
{       fl_tn = fl_lsn/30 ;
        fl_sn = fl_lsn%30 ;    /* offset into cylinder */
        flhead = fl_sn/15 ;
        if (fl_sn >= 15)
                fl_sn -= 15 ;
}
```

```
/*
 * blckxfer is called to start a transfer and
 * calculate if byte count crosses a track boundary.
 */

blckxfer(bp)
struct buf *bp;
{
        unsigned short limit_cnt;

        if (blckcommand(SENS_DR) == FL_ERROR)
                return(FL_ERROR);

        /* Write protect check; caller must check */
        if ((fl_stats[0]&WR_PROT)
                && !(bp->b_flags & B_READ))
        {
                cmn_err(CE_ERROR,
                        "%s: disk is write protected",
                        fl_name);
                return(WP_ERROR);
        }

        fl_eot = fl_sn + flbtos(flresidcnt);
        fltranscnt = flresidcnt;
        if ( fl_eot > 15 )
        {       fl_eot = 15;
                fltranscnt = flstob(15 - fl_sn);
        }

/*
 * Keep transfer from crossing 64K boundary.
 * transfer up to the boundary, blcklocal() the
 * 512 which cross it, then pick it up again.
 */

limit_cnt = -fl_ma;
if (limit_cnt < fltranscnt && limit_cnt) {
        if (limit_cnt < FLSECSZ)
                return(blcklocal(bp));
        else
                fltranscnt = limit_cnt & 0xFE00;
        }

        if(blck_doIO(bp) == FL_ERROR){
                return(FL_ERROR);
        }
        return(0);
}
```

```
/*
 * blck_dma(), a DMA d_proc routine
 * that actually does the DMA transfer
 */

blck_dma()
{
    dma_param  ( dma_q.d_chan, dma_q.d_mode,
                 dma_q.d_addr, dma_q.d_cnt   );
    dma_enable ( dma_q.d_chan );
}

/*
 * blck_doIO does the DMA transfer, setting up the DMA
 * and then issuing the controller command (read/write)
 */

blck_doIO(bp)
struct buf *bp;
{
        if(bp->b_flags&B_READ)          /* Read or write */
        {
                dma_q.d_chan = DMA_CH2;
                dma_q.d_mode = DMA_Rdmode;
                dma_q.d_addr = fl_ma;
                dma_q.d_cnt  = (long) (fltranscnt -1);
                dma_q.d_proc = blck_dma;

                blck_dma();
                if(blckcommand(RD_DATA) == FL_ERROR)
                        return(FL_ERROR);

        }
        else /* write protect was checked in blckstrategy */
        {
                dma_q.d_chan = DMA_CH2;
                dma_q.d_mode = DMA_Wrmode;
                dma_q.d_addr = fl_ma;
                dma_q.d_cnt  = (long) (fltranscnt -1);
                dma_q.d_proc = blck_dma;

                blck_dma();
                if(blckcommand(WR_DATA) == FL_ERROR)
                        return(FL_ERROR);
        }
        return(0);
}
```

```
/*
 * blckcommand() - send a command to the controller.
 * The fl_cmds array in the header file describes the control
 * bytes written for a command and how many bytes are read
 * to find the result. If the command has no execution
 * phase, for example, sense drive status, then the results
 * are read here; otherwise, a sense interrupt status
 * command is issued in blckintr()
 */

blckcommand(func)
unsigned int func;
{
        struct buf *bp;
        unsigned int errcnt, cmdcnt, num_rd,
                        num_wr, execflag, rdcnt;

        bp = blcktab.b_actf;
        fl_job++;           /* for motor off/error timeout */
        switch( func )
        {
                case WR_DATA:
                case RD_DATA: /* set up blckcommand code */
                        blckioset();
                        break;
                case SEEK:
                        blckseekset();
                        break;
                case SENS_INTR:
                        blckISset();
                        break;
                case SENS_DR:
                        blckDSset();
                        break;
                case SPECIFY:
                        blckspecset();
                        break;
                case RECAL:
                        blckrecalset();
                        break;
                default:
                        break;
        }
```

(Continued on next page.)

```
        errcnt = cmdcnt = 0;
        num_wr = fl_cmd[func][NUM_WR];
        num_rd = fl_cmd[func][NUM_RD];
        execflag = fl_cmd[func][EXEC];

        if(IOshake() == FL_ERROR){
                return(FL_ERROR);
        }

        /* output blckcommand parameters */
        while(cmdcnt < num_wr)
        {       if(WRshake() == FL_ERROR)
                        ++errcnt;
                outb(P_FLDCR, fl_cmds[cmdcnt++]);
        }
        if( execflag == FALSE) /* If no execution phase */
        {       rdcnt = 0;
                while( rdcnt < num_rd )
                {       if(RDshake() == FL_ERROR)
                                ++errcnt;
                        fl_stats[rdcnt++] = inb(P_FLDCR);
                }
        }
        if(errcnt){
                return(FL_ERROR);
        }
        return(0);
}
```

```
/*
 * cmdresult(): do the handshaking to read the
 * results of a read or write command
 */

cmdresult(func)
int func;
{
        struct buf *bp;
        int num_rd;
        int cmdcnt, i;

        cmdcnt = 0;
        bp = blcktab.b_actf;

        num_rd = fl_cmd[func][NUM_RD];

        while(cmdcnt < num_rd)          /* Do the reads */
        {       if(RDshake() == FL_ERROR)
                        return(FL_ERROR);
                fl_stats[cmdcnt++] = inb(P_FLDCR);

                /* arbitrary delay loop */
                for (i = 0 ; i < 5 ; i++ );

                if(!inb(P_FLMSR)&CB)
                        break;
        }
        return(0);
}
```

```
679   /*******************************************************
680    * The following routines are called from blckcommand()
681    * to set up the fl_cmds[] array (line 92) with the
682    * sequence of bytes for that command.  blckcommand()
683    * does the actual work of writing to the controller.
684    */

685   blckioset()
686   {
687           struct buf *bp;

688           bp = blcktab.b_actf;
689           fl_cmds[0] = ((bp->b_flags&B_READ)
690                        ? (RD_DATA|MF|SK) : (WR_DATA|MF));
691           fl_cmds[1] = 0 | (flhead << 2);
692           fl_cmds[2] = fl_tn;
693           fl_cmds[3] = flhead ;
694           fl_cmds[4] = fl_sn + 1;
695           fl_cmds[5] = flparam.fl_nbs;
696           fl_cmds[6] = fl_eot;
697           fl_cmds[7] = flparam.fl_gpl;
698           fl_cmds[8] = flparam.fl_dtl;
699   }

700   blckDSset()
701   {
702           fl_cmds[0] = SENS_DR;
703           fl_cmds[1] = 0 ;
704   }

705   blckISset()
706   {
707           fl_cmds[0] = SENS_INTR;
708   }

709   blckrecalset()
710   {
711           fl_cmds[0] = RECAL;
712           fl_cmds[1] = 0;
713           recalstat = 1;    /* set bit saying this   */
714   }                         /* drive is recalibrated */

715   blckseekset()
716   {
717           fl_cmds[0] = SEEK;
718           fl_cmds[1] = 0 ;
719           fl_cmds[2] = fl_tn;
720   }
```

```
blckspecset()
{
        fl_cmds[0] = SPECIFY;
        fl_cmds[1] = flparam.fl_spc1;
        fl_cmds[2] = flparam.fl_spc2;
}

/*
 * Raw interface routines
 */

blckbreakup(bp)
struct buf *bp;
{
        dma_breakup(blckstrategy, bp);
}

blckread(dev)
dev_t dev;
{
    if (physck((daddr_t) 2400, B_READ))
        physio(blckbreakup, &rblckbuf, dev, B_READ);
}

blckwrite(dev)
dev_t dev;
{
    if (physck((daddr_t) 2400, B_WRITE))
        physio(blckbreakup, &rblckbuf, dev, B_WRITE);
}
```

```
/*******************************************************
 * Miscellaneous utility routines
 */


/*
 * cmd_status() - interpret returned status register 0
 */

cmd_status()
{
        register int ic, us;

        ic = (fl_stats[0] >> 6) & 0x03; /* interrupt code */
        us = fl_stats[0] & 0x03;        /* unit select    */

        switch(ic)
        {        case 0:
                        return(0);
                case 1:
                case 2:
                        return(FL_ERROR);
                case 3:
                        if ((us == 0)
                                || (flstate == STRESET))
                                return(FL_CHANGE);
                        break;
                default:
                        break;
        }
        return(0);
}

/* handshake routines for reading */
/* and writing the data registers */

WRshake()
{
        unsigned int status;
        unsigned short timeout;

        timeout = 0;
        while(((status = inb(P_FLMSR))&(RQM|DIO)) != RQM )
        {        if(—timeout == 0)
                        return(FL_ERROR);
        }
        return(0);
}
```

```
RDshake()
{
        unsigned int status;
        long timeout;

        timeout = 0x4FFFF;
        while(((status = inb(P_FLMSR))&RQM) != RQM )
        {
                if(--timeout == 0)
                {
                        return(FL_ERROR);}
                }
        return(0);
}


IOshake()        /* Wait for controller not busy */
{
        unsigned short timeout;
        timeout = 0;
        while( inb(P_FLMSR)&CB)
        {       if(--timeout == 0)
                        return(FL_ERROR);
        }
        return(0);
}

/*
 * blckreset() - send a reset command.  This occurs
 * both during initialization and for errors.  In
 * the latter case, we should attempt to preserve
 * motor on and select bits, rather than sending a 0.
 */

blckreset()
{
    dor = 0 | FL_ENB_DI;
    flstate = STRESET;
    recalstat = 0;        /* drive must be recalibrated */
    CNTRL_WR(dor);        /* reset */
    dor |= FDC_ENB;        /* clear reset, and send out */
    CNTRL_WR(dor);
    return;
}
```

```
/*
 * motoron() - if the motor of flunit is already
 * on, return 1.  Otherwise turn it on, and
 * return 0.  There is no additional head load wait.
 */

motoron()
{
        ushort delay;

        dor &= ~0x03;           /* mask the drive select bits */
        switch (0) {
        case 0:
                delay = dor & 0x10;
                dor |= 0x1C;
                break;
        default:
                break;
        }
        CNTRL_WR(dor);
        return (delay);
}
```

```
/*
 * blck_timeout() - check to see if a motor should be turned
 * off, or if an operation has hung. The fl_job variable
 * gets incremented in blckcommand().  This routine is
 * called only by timeout() calls in strategy(), intr(),
 * or from blck_timeout itself.  This routine provides
 * the only way to detect that a floppy drive door is open;
 * a timeout occurs because an interrupt never happens.
 */
blck_timeout(t)
{
    int pri;
    register struct buf *bp;
    static ofl_job = 0;
    int mtr;
    bp = blcktab.b_actf;        /* present bp */
    if( ofl_job == fl_job )
    {
        if( t < flparam.fl_moft )
            timeout(blck_timeout, ++t, Hz);
        else {
            fl_to = 0;
            pri = spl5();
            if (bp && blcktab.b_active) {
                cmn_err(CE_CONT,
                  "%s: insert disk or close floppy door\n",
                                                fl_name);
                bp->b_flags |= B_ERROR;
                blcktab.b_errcnt= 0;
                bp->b_resid = 0;
                blcktab.b_active = 0;
                blcktab.b_actf = bp->av_forw;
                iodone(bp);
                flerreset = TRUE ;
                blckstart() ;
                splx(pri);
                return;
            }
            splx(pri);
            /* turn motors off */
            mtr = 0x10;
            dor &= ~mtr;
            CNTRL_WR(dor);
            return;
        }
    }
    else
    {
        ofl_job = fl_job;
        timeout(blck_timeout,0,Hz);
    }
}
```

```
/*
 * blcklocal - do 'local' transfer - if the I/O requested
 * crosses a 64k boundary, the DMA wraps.  Transfer up to
 * the offensive sector and then DMA from/to kernel data
 * and copy in/out to user.  After this, the transfer is
 * picked up as usual. This routine doubles as one which
 * sets up the transfer and the interrupt handler for such
 * transfers, although in the interrupt case, errors have
 * already been checked by the main interrupt handler.
 */

blcklocal(bp)
register struct buf *bp;
{
        if (fllocaltransfer)  {
                /* if so, its interrupt time */
                fllocaltransfer = 0;
                if (bp->b_flags & B_READ)    /* copy out? */
                {
                        copyio(fltmpaddr, (caddr_t)fllocalbuf,
                                FLSECSZ, U_WKD);
                }
                fl_ma = fltmpaddr + FLSECSZ; /* reset vars*/
                flresidcnt -= FLSECSZ;
                fltrnsferred += FLSECSZ;
                fl_lsn += 1;
                /* done with local, finish    */
                /* with transfer like normal  */
                if (flresidcnt == 0)
                        goto localdone;
                else {
                        blckmap();
                        flstate = STSEEK;
                        if (blckcommand(SEEK) == FL_ERROR)
                                goto localerror;
                        return;
                }
 localerror:
                if (++ blcktab.b_errcnt < MAXRETRY)
                {
                        flstate = STRESET ;
                        blckreset();
                        return ;
                }
                /* hard error */
                bp->b_flags |= B_ERROR ;
                flerreset = TRUE ;
                /* will have to do reset before next xfer */
```

(*Continued on next page.*)

```
localdone:
                blcktab.b_actf = bp->av_forw ;
                blcktab.b_errcnt = 0 ;
                blcktab.b_active = 0 ;
                bp->b_resid = bp->b_bcount - fltrnsferred ;
                iodone(bp) ;
                blckstart() ;
        } else {                /* set up transfer */
                /* remember real address */
                fltmpaddr = fl_ma;
                /* DMA into kernel data */
                fl_ma = ktop(fllocalbuf);
                if ((bp->b_flags
                         & B_READ) == 0) /* write? */
                {
                     copyio(fltmpaddr, (caddr_t)fllocalbuf,
                            FLSECSZ, U_RKD);
                }
                fltranscnt = FLSECSZ;  /* just one sector */
                /* flag for interrupt time */
                fllocaltransfer = 1;
                if (blck_doIO(bp) == FL_ERROR)   /* do it */
                        return(FL_ERROR);
                return(0);
        }
}

/*
 * blckhalt - called from dhalt() at system shutdown.
 */

blckhalt()
{
        CNTRL_WR(FDC_ENB);  /* motors off, de-select, */
                            /* disable DMA/interrupts */
}
```

```
/* queue the request to the blckstart() routine. */
/* start the device if necessary.                */

devque (bp)
register struct buf *bp;
{
    register dev, lev;

    dev = bp->b_dev;
        bp->b_sector = bp->b_blkno;
    bp->b_flags &= ~B_DONE;      /* reset done flag */
    lev = spl5();
    disksort(&blcktab, bp);
    while ( blcktab.b_active == 0 )
    {
        dma_alloc ( DMA_CH2 , DMA_BLOCK );
        if ( blcktab.b_active == 0  ) {
            blckstart();
            break;
        }
    }
    splx (lev);
}


blckprint(dev, str)
dev_t dev;
char *str;
{
        cmn_err(CE_NOTE,
                "%s on floppy diskette unit %d, minor %d",
                str, unitbits(dev), minor(dev));
}

/*
** flopen - provided for call from kernel
**
*/
flopen(dev, mode, flag)
{
        blckopen(dev, mode, flag);
}
```

Appendix C

Section (K) Manual Pages

Manual Page Overview C-1

Manual Page Overview

The section (K) routines listed in this appendix are provided in the kernel for writing a device driver.

Contents

Kernel Routines **(K)**

Intro	lists manual page references
bcopy	copies bytes in kernel space
brelse	releases a block buffer
btoc, btoms, ctob	converts between bytes and clicks (memory pages)
bzero	sets memory locations to 0 (zero)
canon	processes raw input data from tty device
cmn_err	displays message or panics the system
copyin, copyout	copies bytes between user and kernel space
copyio	copies bytes to and from a physical address
cpass, passc	passes character between user space and the kernel
db_alloc, db_free	allocates and frees physically contiguous memory
db_read	transfers data from physical memory to a user address
db_write	transfers from a user address to contiguous memory
delay	delays process execution for specified time
deverr	prints a device error message on the console
disksort	adds a block I/O request to a device's queue
dma_alloc	allocates a DMA channel
dma_breakup	sizes DMA request into 512-byte blocks
dma_enable	begins DMA transfer
dma_param	sets up a DMA controller chip for DMA transfer
dma_relse	releases previously allocated DMA channel
dma_resid	returns the number of bytes not transferred during a DMA request
dma_start	queues DMA request
emdupmap	duplicates channel mapping
emunmap	disables mapping on a channel
flushtlb	flushes the translation lookaside buffer
fubyte	gets a character from user data space
fuword	gets one 32-bit word from user data space
getc, getcb, getcbp, getcf	read clist buffers
getchar	gets one character of input
geteblk, getablk	gets a buffer from the block buffer pool
inb, outb	reads a byte from or writes a byte to an I/O address
ind, outd	reads, writes a 32-bit word to a physical I/O address
inw, outw	reads, writes a 16-bit word from or to a physical I/O address

iodone	signals I/O completion
iowait	wait for I/O completion
longjmp	ends current system call with error
major, makedev, minor	returns major, new device number, or minor device number
memget	allocates contiguous memory at initialization
paddr	returns virtual address pointer to block data
panic	halts the system
physio, physck	raw I/O for block drivers
pio_breakup	breaks up programmed I/O requests
printcfg	displays driver initialization message
printf	print a message on the console
psignal	sends signal to a process
ptok, ktop	converts virtual and physical addresses
putc, putcb, putcbp, putcf	write to clists
putchar	prints a character on the console
repins: repinsb, repinsw, repinsd, repoutsb, repoutsw, repoutsd	reads and writes streams of device data
scsi: scsi_get_gen_cmd, scsi_getdev, scsi_mkadr3, scsi_s2tos, scsi_s3tol, scsi_stok, scsi_stol, scsi_swap4	SCSI routines
select: selsuccess, selfailure, selwakeup	kernel routines supporting select(S)
seterror	sets u.u_error with error code
signal	sends a signal to a process
sleep	suspends processing temporarily
spl: spl0, spl1, spl2, spl3, spl4, spl5, spl6, spl7, splbuf, splcli, splhi, splni, splpp, spltty, splx	block/permit interrupts

sptalloc	allocates temporary memory or maps a device into memory
sptfree	releases memory previously allocated with sptalloc
subyte	stores a character in user data space
suser	determines if current user is the super-user
suword	stores a 32-bit word in user data space
timeout, untimeout	schedules a time to execute a routine
ttiocom	interpret tty driver I/O control commands
tty: ttclose, ttin, ttinit, ttiwake, ttopen, ttout, ttowake, ttread, ttrdchk, ttrstrt, ttselect, tttimeo, ttwrite, ttxput, ttyflush, ttywait	tty driver routines
vas: vasbind, vasmalloc, vasmapped, vasunbind	virtual address space memory routines
video: DISPLAYED, viddoio, vidinitscreen, vidmap, vidresscreen, vidsavscreen, vidumapinit, vidunmap	supports video adapter driver development
vtop	convert a virtual address to a physical address
wakeup	wakes up a sleeping process

Intro

lists manual page references

Description

This section describes the manual page on which each kernel routine is found.

The following table summarizes the kernel routines. The columns indicate the routine name, the manual page on which the routine appears, a description, and a code that indicates the following values:

Letter	Meaning
B	Use this routine only in a block driver
C	Use this routine only in a character driver
G	General; can be used in a block or character driver
I	Routine can be called from an initialization routine
X	Routine can be called from an interrupt routine
M	Macro
A	Written in Assembly language

The section (K) routines are summarized in the following table:

Kernel Routine	Manual Page	Description	Code
bcopy	**bcopy**	Copies bytes in kernel space	GAIX
brelse	**brelse**	Releases a block buffer	B
btoc	**btoc**	Returns number of pages (clicks)	GMIX
btoms	**btoc**	Returns number of sectors	GMIX
bzero	**bzero**	Sets memory locations to 0 (zero)	GAIX
canon	**canon**	Processes raw input data from tty device	CA
cmn_err	**cmn_err**	Displays message or panics the system	GIX
copyin	**copyin**	Copies bytes from user to kernel space	GA
copyio	**copyio**	Copies bytes to/from physical address	G
copyout	**copyin**	Copies bytes from kernel to user space	GA
cpass	**cpass**	Passes character to user	G
ctob	**btoc**	Returns number of bytes	GMIX

Kernel Routine	Manual Page	Description	Code
db_alloc	**db_alloc**	Allocates physically contiguous memory	GI
db_free	**db_alloc**	Frees physically contiguous memory	GI
db_read	**db_read**	Transfers data from kernel virtual to physical address	G
db_write	**db_write**	Transfers data from physical to kernel virtual address	G
delay	**delay**	Delays process execution for specified time	G
deverr	**deverr**	Prints message on console	B
disksort	**disksort**	Adds block I/O request to device's queue	BX
DISPLAYED	**video**	Returns TRUE if screen displayed	CIX
dma_alloc	**dma_alloc**	Allocates a DMA channel	GIX
dma_breakup	**dma_breakup**	Sizes request into 512-byte blocks	B
dma_enable	**dma_enable**	Begins DMA transfer	GIX
dma_param	**dma_param**	Sets up a DMA controller chip for DMA transfer	GIX
dma_relse	**dma_relse**	Releases previously allocated DMA channel	GIX
dma_resid	**dma_resid**	Returns bytes not transferred for DMA request	G
dma_start	**dma_start**	Begins DMA transfer	G
emdupmap	**emdupmap**	Duplicates channel mapping	G
emunmap	**emunmap**	Disables mapping on a channel	G
flushtlb	**flushtlb**	Flushes the translate lookaside buffer	GAIX
fubyte	**fubyte**	Gets a character from user data space	GA
fuword	**fuword**	Gets a 32-bit word from user data space	GA
getablk	**geteblk**	Gets empty buffer from free list	B
getc	**getc**	Gets a character from a clist	C
getcb	**getc**	Gets cblock from clist	C
getcbp	**getc**	Gets characters from a clist	C
getcf	**getc**	Gets a cblock from free list	C
getchar	**getchar**	Gets one character of input	G
geteblk	**geteblk**	Gets empty buffer from free list	B
inb	**inb**	Reads a byte from an I/O address	GAIX
ind	**ind**	Reads double words from an I/O address	GAIX

Kernel Routine	Manual Page	Description	Code
inw	**inw**	Reads a 16-bit word from a physical I/O address	GAIX
iodone	**iodone**	Signals I/O completion	BIX
iowait	**iowait**	Wait for I/O completion	B
ktop	**ptok**	Returns physical address from kernel	GMIX
longjmp	**longjmp**	Restores previously saved process context	GA
major	**major**	Returns major number from the device number	GM
makedev	**major**	Returns device number from major and minor numbers	GM
memget	**memget**	Allocates contiguous memory at initialization	GI
minor	**major**	Returns minor device number from the device number	GM
outb	**inb**	Writes a byte to an I/O address	GAIX
outd	**ind**	Writes double words to a physical I/O address	GAIX
outw	**inw**	Writes a 16-bit word to a physical I/O address	GAIX
paddr	**paddr**	Returns virtual address pointer to block data	BIX
panic	**panic**	Halts the system	GIX
passc	**cpass**	Passes character between user space and the kernel	G
physck	**physio**	Verifies I/O request size	B
physio	**physio**	Performs physical I/O	B
pio_breakup	**pio_breakup**	Breaks up programmed I/O requests	B
printcfg	**printcfg**	Displays driver initialization message	GI
printf	**printf**	Print a message on the console	GIX
psignal	**psignal**	Sends signal to a process	CIX
ptok	**ptok**	Returns kernel address from physical	GMIX
putc	**putc**	Puts character on clist	C
putcbp	**putc**	Puts characters on clist	C
putcb	**putc**	Puts cblock on clist	C
putcf	**putc**	Puts cblock on free list	C
putchar	**putchar**	Prints a character on the console	G
repinsb	**repinsb**	Moves bytes to memory from I/O address	GA

Kernel Routine	Manual Page	Description	Code
repinsd	**repinsb**	Moves words to memory from I/O address	GA
repinsw	**repinsb**	Moves double words to memory from I/O address	GA
repoutsb	**repinsb**	Moves bytes from memory to I/O address	GA
repoutsd	**repinsb**	Moves words from memory to I/O address	GA
repoutsw	**repinsb**	Moves double words from memory to I/O address	GA
scsi_get_gen_cmd	scsi	Fills a command block	GIX
scsi_getdev	scsi	Gets SCSI device number	GIX
scsi_mkadr3	scsi	Makes 3-byte address	GIX
scsi_s2tos	scsi	Converts 2 bytes to short	GIX
scsi_s3tol	scsi	Converts 3 bytes to long	GIX
scsi_stok	scsi	Converts 3 bytes to address	GIX
scsi_swap4	scsi	Swaps 4 bytes	GIX
selfailure	**select**	Fails condition	G
selsuccess	**select**	Okays condition	G
selwakeup	**select**	Okays failed condition	G
seterror	**seterror**	Sets u.u_error with error code	G
signal	**signal**	Sends a signal to a process	CIX
scsi_stol	scsi	Converts 4 bytes to long	GIX
sleep	**sleep**	Suspends processing temporarily	G
spl0	**spl**	Permits all interrupts	GAIX
spl1	**spl**	Blocks context switch interrupts	GAIX
spl2	**spl**	Blocks level 2 interrupts	GAIX
spl3	**spl**	Blocks level 3 interrupts	GAIX
spl4	**spl**	Blocks level 4 interrupts	GAIX
spl5	**spl**	Blocks block device interrupts	GAIX
spl6	**spl**	Blocks character device and the clock's interrupts	GAIX
spl7	**spl**	Blocks all interrupts	GAIX
splbuf	**spl**	Blocks buf access interrupts	GAIX
splcli	**spl**	Blocks clist access interrupts	GAIX
splhi	**spl**	Blocks all interrupts	GAIX
splni	**spl**	Blocks network interrupts	GAIX
splpp	**spl**	Blocks ports board interrupts	GAIX
spltty	**spl**	Blocks tty interrupts	GAIX
splx	**spl**	Enables previous spl level	GAIX
sptalloc	**sptalloc**	Allocates temporary memory or maps a device into memory	GI
sptfree	**sptfree**	Releases memory previously allocated with sptalloc	GI

Kernel Routine	Manual Page	Description	Code
subyte	**subyte**	Stores a character in user data space	GA
suser	**suser**	Determines if current user is the super-user	G
suword	**suword**	Stores a 32-bit word in user data space	GA
timeout	**timeout**	Schedules a time to execute a routine	GX
ttclose	**tty**	Closes access to tty device	C
ttin	**tty**	Gets data from receive buffer	CX
ttinit	**tty**	Initializes line discipline	C
ttiocom	**ttiocom**	Interpret tty driver I/O control commands	C
ttiwake	**tty**	Awakens input requests	CX
ttopen	**tty**	Opens tty device	C
ttout	**tty**	Puts data into transmit buffer	CX
ttowake	**tty**	Awakens output requests	CX
ttrdchk	**tty**	Verifies characters to read	C
ttread	**tty**	Copies tty data to user space	C
ttrstrt	**tty**	Restarts tty access	C
ttselect	**tty**	Ensures read or write without block	C
tttimeo	**tty**	Times input request	C
ttwrite	**tty**	Copies data from user	C
ttxput	**tty**	Puts data into output queue	C
ttyflush	**tty**	Releases queue contents	C
ttywait	**tty**	Waits for UART to drain	C
untimeout	**timeout**	Cancel scheduled timeout request	GX
vasbind	**vas**	Binds virtual address to physical	GI
vasmalloc	**vas**	Allocates virtual user memory	GI
vasmapped	**vas**	Releases allocated memory	GI
vasunbind	**vas**	Unbinds bound memory	GI
viddoio	**video**	Supports I/O controls	C
vidinitscreen	**video**	Initializes multiscreen	CI
vidmap	**video**	Maps memory	CIX
vidresscreen	**video**	Restores screen	CX
vidsavscreen	**video**	Saves screen	CX
vidumapinit	**video**	Maps user memory	CI
vidunmap	**video**	Unmaps memory	CIX
vtop	**vtop**	Convert a virtual address to a physical address	GX
wakeup	**wakeup**	Wakes up a sleeping process	GX

bcopy

copies bytes in kernel space

Syntax

```
int
bcopy(src, dst, cnt)
caddr_t src, dst;
int cnt;
```

Description

The argument *src* is a pointer to the kernel address the data is transferred from. The argument *dst* is a pointer to the kernel address the data is transferred to. If the destination address is outside of kernel space, the system panics. The value of *cnt* is the number of bytes to transfer.

Do not use for moving data to user space; use *copyin*, *copyio*, or *copyout* instead.

See Also

copyin(K), copyio(K), copyout(K)

brelse

releases a block buffer

Syntax

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

Description

The *brelse* routine releases a block buffer to the free pool of buffers. This routine is called by a block device driver to release a buffer. The contents of the buffer are lost and the driver is not allowed to make any further reference to the buffer. This routine is called by *iodone*(K) at the completion of a block I/O request.

When the routine is first called, **bp->b_flags** is checked for an error (B_ERROR is set). If it is, the following occurs:

- The B_STALE flag is set in **bp->b_flags**.
- The B_ERROR and B_DELWRI flags are removed from **bp->b_flags**.
- The **bp->b_error** field is set to zero.

All processes asssociated with *bp* that are sleeping waiting for a buffer header or a free buffer are awakened. On completion of *brelse*, **b_proc** is set to zero to release the process's ownership of the buffer.

Parameters

The *bp* argument is a pointer to the buffer header relating to the buffer to be released.

Return Value

The buffer addressed by *bp* is returned to the free buffer pool. No errors are possible.

Notes

Note that this routine can only be called from block device drivers.

btoc, btoms, ctob

converts between bytes and clicks (memory pages)

Syntax

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

unsigned
btoc(bytes)
unsigned bytes;

unsigned
btoms(bytes)
unsigned bytes;

unsigned
ctob(clicks)
unsigned clicks;
```

Description

The *btoc* and *ctob* macros convert between bytes and clicks (memory pages). *btoms* is an alias for *btoc*. *btoc* (or *btoms*) returns the number of memory pages that are needed to contain the specified number of bytes. For example, if the page size is 4096 bytes, then *btoc(5000)* returns 2.

ctob returns the number of bytes contained in the specified memory pages. For example, if the page size is 4096 bytes, then *ctob(2)* returns 8192.

btoc(0), *btoms(0)*, or *ctob(0)* each return 0 (zero).

bzero

sets memory locations to 0 (zero)

Syntax

```
int
bzero(address, bytes)
caddr_t address;
int bytes;
```

Description

This routine clears a contiguous portion of memory by filling the memory with zeros. *address* is an even-word address specifying the beginning of the area to clear. *bytes* is an even-word value specifying the number of bytes to clear.

canon

processes raw input data from tty device

Syntax

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

int
canon(tp)
struct tty *tp;

Description

canon is called by *ttread*(K) to process characters received from a tty device, indicated by the *tp* argument. The *canon* routine conveys characters from the raw input queue, the **t_rawq** field, to the processed character queue, the **t_canq** field. The *canon* routine receives characters until a delimiter is encountered in the input data. (All **t_** fields shown on this manual page are members of the **tty** structure described in **sys/tty.h**.)

When the delimiter is found, then the accumulated characters are processed and sent to the calling program. The **t_delct** delimiter count field indicates that a delimiter character has been received. Until a delimiter is received, *canon* can call *sleep*(K) to wait for characters to be placed in the raw queue. The priority argument to *sleep* is TTIPRI.

canon processes characters as long as there is a carrier and as long as FNDELAY (no-delay mode) is not set.

The *canon* routine must not be called from a driver's initialization or interrupt routines.

The *canon* routine has two character processing modes. The first mode is called the canonical processing mode. Canonical processing means resolving special characters such as backspace or delete before the received data is given to the calling program. The *termio*(M) manual page describes guidelines that are used when canonical processing takes place. Refer to the description of the **c_cc** array on the *termio* manual page for information about which characters are resolved by canonical processing. Canonical processing mode is enabled by setting the ICANON flag in the **t_lflag** field. In canonical processing mode, international characters are translated, as are these *termio* constants: VERASE, VKILL, VEOF, VEOL, VEOL2, and XCASE.

The second mode passes characters to a calling program after a time requirement has been satisfied or after a minimum number of characters have been received. This mode requires that ICANON not be set and that the **t_cc[VMIN]** field be set to zero. This mode interacts with the VMIN and VTIME constants which are described on the *termio* manual page. After the time and minimum character requirements are satisfied, characters are conveyed from **t_rawq** to **t_canq**. *tttimeo*(K) is called in this mode to resolve VTIME.

If during the course of either processing modes, as long as characters remain to be processed, if **t_state** is set to TBLOCK, then the *canon* routine calls the driver's *xxproc* routine with the T_UNBLOCK argument.

Notes

The *canon* routine must not be called from a driver's initialization or interrupt routines.

Parameters

tp	a pointer to the *struct tty* data structure associated with the device being accessed.

See Also

termio(M), ttiocom(K), tty(K)

cmn_err

displays message or panics the system

Syntax

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

int
cmn_err(severity, format, arguments)
char *format;
int severity, arguments;
```

Description

cmn_err displays a message on the console and can additionally, panic the system. This routine is an improvement over *printf* in that *cmn_err* automatically handles a variety of terminals such as bit-mapped graphics terminals (which *printf* cannot handle), and *cmn_err* permits grading messages into four categories by level of severity. The levels are for continued messages, notice messges, warning messages, and panic messages. If a panic message is called immediately after another panic request, a ‘‘double panic’’ results.

cmn_err is also used to store messages in *putbuf*, a circular array in memory that can be accessed from *crash*(ADM). All messages are also stored in **/usr/adm/messages** (an error background program conveys the messages to this file).

Warning

An incorrect *severity* argument value causes a system panic. Also, this routine is not interrupt-driven and therefore suspends all other system activities while executing.

Parameters

severity	One of four different levels for indicating the severity of the message. Use CE_NOTE to display a message preceded with NOTICE:. Use CE_WARN to display a message preceded with WARNING:. Use CE_PANIC to panic the system and to display a message preceded with PANIC:. If two or more panic requests are called at the same time, DOUBLE PANIC: is displayed. Use CE_CONT to continue a previous message or to

display a message without a severity indicator preceding the message. The *severity* argument must be specified; if omitted, a panic results.

format A message to be displayed. The destination of the message is specified with the first character of *format*. If the first character is an exclamation point (!), then the message is stored in *putbuf* and not displayed on the console. If the first character is a carat (^), then the message is displayed on the console and not stored in *putbuf*. Any other character at the start of *format* sends the message to both *putbuf* and the console. The message is appended with a *\n* for all *severity* levels except for CE_CONT. *format* accepts data type specifications the same as *printf*(K) for displaying the arguments passed to the message string. The supported specifications are:

Type	**Description**
%b	two-digit hexadecimal byte
%c	character
%d	signed decimal
%o	unsigned octal
%s	string (character pointer)
%x	hexadecimal (prints leading zeros)

Specifications can be indicated in either upper or lower case. Field length specifications cannot be used for arguments. For example, *%9d* is not permitted. Escaped characters such as *\n*, *\t*, *\033*, and so on are C language features that are supported by the C compiler and are thus supported in this kernel routine.

arguments optional variables to be displayed using the *format* argument.

Example

You can use *cmn_err* to display messages on the console as follows:

```
cmn_err(CE_NOTE," xxioctl routine called - device number = %x",
        device);
cmn_err(CE_CONT," this is not a problem.\n");
```

copyin, copyout

copies bytes between user and kernel space

Syntax

```
int
copyin(src, dst, cnt)
caddr_t src;
caddr_t dst;
int cnt;

int
copyout(src, dst, cnt)
caddr_t src;
caddr_t dst;
int cnt;
```

Description

The *copyin* routine copies bytes from user space to kernel space. The *copyout* routine copies bytes from kernel space to user space. After completion of calls to these routines, increase **u.u_base** by the number of bytes transferred, and decrease **u.u_count** by the number of bytes transferred. If an error code is returned, call *seterror(EFAULT)* to return EFAULT to the user process that is calling your driver. Because these routines access a user process, neither can be used in an interrupt or initialization routine. The driver is not required to supply word-align addresses to these routines.

Parameters

For *copyin*, the argument *src* is a 32-bit pointer that contains the offset of the user address the data is copied from. Often, *src* is obtained from either *u.u_base* or the third argument passed to a driver's *xxioctl* routine (*arg*).

The argument *dst* is a pointer to the kernel address (buffer address) that the data is transferred to.

The argument *cnt* specifies the number of bytes to transfer.

For *copyout*, the argument *src* is a pointer to the kernel address (in the buffer) that the data is transferred from.

The argument *dst* is a 32-bit pointer that contains the offset of the user address the data is copied to.

The argument *cnt* specifies the number of bytes to transfer.

Return Value

If successful, these routines perform the specified data transfer; otherwise, -1 is returned for one of the following reasons:

- A page fault occurred between a transfer to user space.
- The address in user space is invalid.
- An address was specified that would have resulted in data being copied into the user block.

Example

Assuming *arg* is a pointer to a user data structure that was passed via *xxioctl*, use *copyin* to copy from user data space to kernel data space.

```
xxioctl(dev, cmd, arg, mode)
dev_t dev;
int cmd, mode;
caddr_t arg;
{
        struct foo dst;
        .
        .  other ioctl code
        .
        /* copy from arg to dst */
        if ( copyin(arg, &dst, sizeof(struct foo)) == -1)
        {
                u.u_error = EFAULT;
                return;
                    }
```

Assuming *arg* is a pointer to a user's data structure that was passed via *xxioctl*, use *copyout* to copy from kernel data space to user data space.

```
xxioctl(dev, cmd, arg, mode)
dev_t dev;
int cmd, mode;
caddr_t arg;
{
        struct foo dst;
        .
        .  other ioctl code
        .

        /* copy from dst to arg */
        if (copyout(&dst, arg, sizeof(struct foo)) == -1)
        {
                u.u_error = EFAULT;
                return;
        }
        .
        .
}
```

See Also

bcopy(K), copyio(K), seterror(K)

copyio

copies bytes to and from a physical address

Syntax

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

int
copyio(paddr, caddr, bytes, mapping)
paddr_t paddr;
caddr_t caddr;
int bytes, mapping;
```

Description

The *copyio* routine copies bytes between kernel virtual addresses, and between kernel addresses and user addresses. This routine has little purpose other than to call *bcopy*(K) for tranfers between kernel addresses, *copyin*(K) for transfers from user space to the kernel, and *copyout*(K) for transfers from the kernel to user space.

After completion of calls to *copyio* with the U_RUD or U_WUD mappings, increase **u.u_base** by the number of bytes transferred, and decrease **u.u_count** by the number of bytes transferred. If an error code is returned, call *seterror(EFAULT)* to return EFAULT to the user process that is calling your driver. Address values need not be word-aligned.

Parameters

The argument *paddr* is a pointer to a virtual address to which or from which data is to be transferred.

The argument *caddr* is a virtual address to which or from which data is to be transferred.

The argument *bytes* is an integer that specifies the number of bytes of data to transfer.

The value of *mapping* is an integer that designates the direction of the transfer. The following possible mapping values are defined in *sys/user.h*. Use of any other value causes a system panic.

U_RKD	Kernel-to-kernel transfer from *paddr* to *caddr* using *bcopy*
U_WKD	Kernel-to-kernel transfer from *caddr* to *paddr* using *bcopy*
U_RUD	Kernel-to-user transfer from *paddr* to *caddr* using *copyout*
U_WUD	User-to-kernel transfer from *caddr* to *paddr* using *copyin*

Warning

Always include **user.h** and select a correct *mapping* value. Otherwise, a panic occurs with the message, "bad mapping in copyio." Be sure to specify kernel addresses only for U_RKD and U_WKD. If a user address is specified, the system will panic. The *copyio* routine called with the U_RUD and U_WUD *mappings* cannot be used from an interrupt or initialization routine.

Return Value

If successful, this routine performs the specified data transfer; otherwise, -1 is returned for any mapping setting if the requested number of bytes to transfer is 0 (zero). The following errors can also occur only when *copyio* is called with the U_RUD or U_WUD mappings:

- a page fault occurred between a transfer to user space
- the address in user space is invalid
- an address was specified that would have resulted in data being copied into the user block

If U_RUD or U_WUD is set in *mapping* and -1 is returned, call *seterror* to return EFAULT to the caller in user space.

See Also

bcopy(K), copyin(K), copyout(K), seterror(K)

cpass, passc

passes character between user space and the kernel

Syntax

int
cpass( )

int
passc(c)
int c;

Description

The *cpass* routine returns the next character in a user write request. *cpass* calls *fubyte*(K), but provides a more usable interface than *fubyte* in that *cpass* automatically updates **u.u_count**, **u.u_offset,** and **u.u_base**, and returns errors in **u.u_error**. If your data may contain a -1, use **copyin**(K) instead of *cpass* because a -1 in the data causes *cpass* (or *fubyte*) to return an error.

The *passc* routine passes a character to a user read request. *passc* calls *subyte*(K), updates **u.u_count**, **u.u_offset**, and **u.u_base**, and returns errors in **u.u_error**. If your data may contain a -1, use **copyout**(K) instead of *passc* because a -1 in the data causes *passc* (or *subyte*) to return an error.

Neither of these routines can be called from a driver's **xxinit** or interrupt routines.

Parameters

The character *c* is passed to the read request by *passc* .

Return Value

The *cpass* routine returns a character. The -1 value may be returned if no characters remain in the output request in **u.u_count** or if an error occurred when transferring the data from user space. If an error occurred, then EFAULT is set in **u.u_error**.

The *passc* routine returns 0 normally and -1 when the user read request has been satisfied.

See Also

fubyte(K), subyte(K)

db_alloc, db_free

allocates and frees physically contiguous memory

Syntax

```
int
db_alloc(dv)
struct devbuf *dv;

int
db_free(dv)
struct devbuf *dv;
```

Description

The *db_alloc* routine allocates one block of physically contiguous memory. Contiguous memory is necessary for performing DMA transfers. Memory for all other uses should be allocated using standard memory allocation routines for your machine. *dv* points to an instance of the *devbuf* structure. Set the *size* field in the **devbuf** structure to the block size before calling *db_alloc*.

db_free releases the previously allocated memory.

The *devbuf* structure is:

Type	**Field**	**Description**	
paddr_t	bufptr;	/* pointer to start of buffer	*/
paddr_t	bufend;	/* pointer to end of buffer	*/
long	size;	/* size of buffer	*/
paddr_t	head;	/* put buffer data here	*/
paddr_t	tail;	/* get buffer data here	*/

Except for **size**, all other fields in the **devbuf** structure are read-only.

Warning

db_alloc must not be used during the driver's initialization routine. The *memget*(K) routine can be called to obtain contiguous memory during driver initialization. Reading from and writing to memory areas allocated using *db_alloc*(K) must be performed using the *db_read*(K) and *db_write*(K) routines only.

Return Value

For *db_alloc*, zero (0) is returned if no memory is available; otherwise, 1 is returned. *db_free* always returns zero (0). for normal completion.

Examples

The following example allocates a single 120K buffer:

```
struct devbuf dv;
dv.size = (long) (120 * 1024);  /* 120 times 1K */
if (db_alloc(&dv) == 0) {
        cmn_err(CE_NOTE, "db_alloc failed");
        return(-1);
}
```

The following example releases previously allocated memory:

```
struct devbuf dv;
db_free(&dv);
```

See Also

db_read(K), db_write(K)

db_read

transfers data from physical memory to a user address

Syntax

int
db_read(dv, va, count)
struct devbuf *dv;
caddr_t va;
unsigned count;

Description

The *db_read* routine transfers data from physical contiguous memory pointed to by *dv* to a virtual user address *va*. The amount transferred is in *count* bytes. The physical memory must have been allocated by *db_alloc*(K).

Warning

Only use this routine after the requested pages are locked into memory by a previous call to *physio*(K). Use of *db_read* under any other circumstances results in a panic.

Example

For example, to transfer data from *dv* to a user buffer:

```
struct buf *bp;
struct devbuf dv;

db_read( &dv, paddr( bp ), bp->b_count );
```

See Also

db_alloc(K), db_write(K), physio(K)

db_write

transfers from a user address to contiguous memory

Syntax

int
db_write(dv, va, count)
struct devbuf *dv;
paddr_t va;
unsigned count;

Description

The *db_write* routine transfers data from a virtual user address *va* to physical contiguous memory pointed to by *dv*. The amount transferred is in *count* bytes. The physical memory must have been allocated by *db_alloc*(K).

Warning

Only use this routine after the requested pages are locked into memory by a previous call to *physio*(K). Use of *db_write* under any other circumstances results in a panic.

Example

For example, to transfer from a buffer to a *dv*:

```
struct buf *bp;
struct devbuf dv;

db_write( &dv, paddr( bp ), bp->b_count );
```

See Also

db_alloc(K), db_read(K), paddr(K)

delay

delays process execution for specified time

Syntax

```
int
delay(ticks)
int ticks;
```

Description

The *delay* routine uses the *sleep*(K) and *wakeup*(K) calls to delay the current process for the specified number of clock ticks.

Parameters

ticks is an integer that specifies the number of clock ticks to delay.

Return Value

After the specified time, the delayed routine resumes running. No value is returned.

Warning

delay cannot be called from an *xxinit* or interrupt routine.

See Also

timeout(K), sleep(K), wakeup(K)

deverr

prints a device error message on the console

Syntax

#include "sys/cmn_err.h"

int
deverr(iobuf-ptr, cmd, status, dev)
struct iobuf *iobuf-ptr;
int cmd, status;
char *dev;

Description

The *deverr* routine prints an error message on the system console together with some device-specific information acquired from the parameters passed to the routine. This routine can only be used in a block device driver.

deverr utilizes the following display call:

```
cmn_err(CE_WARN, "error on dev %s (%u/%u), block=%D cmd=%x
                 status=%x\n", dev, major(bp->b_dev), minor(bp->b_dev),
                 bp->b_blkno, cmd, status);
```

bp is defined as follows: `bp=`*iobuf-ptr*`->b_actf`

This produces a warning message in the following format:

```
WARNING: error on dev dev (major/minor), block=blk
         cmd=cmd status=status
```

Where:

dev	The *dev* (device name) argument to *deverr*
major	Major device number
minor	Minor device number
blk	Block number
cmd	The *cmd* argument to *deverr*
status	The *status* argument to *deverr*

Parameters

The *iobuf-ptr* argument is a pointer to the head of the I/O request queue for the device.

The *cmd* argument contains driver-specific information, such as the controller information from the failed I/O operation.

The *status* argument contains driver-specific information, such as the controller status information from the time of failure.

The *dev* argument is a pointer to a string containing the device name.

See Also

cmn_err(K), printf(K)

disksort

adds a block I/O request to a device's queue

Syntax

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

int
disksort(xxtab, bp)
struct iobuf *xxtab;
struct buf *bp;
```

Description

The *disksort* routine adds a block device I/O request to the queue of such requests for a particular device. The device *xxstrategy* routine normally calls *disksort*. The *xxtab* parameter points to the head of the request queue, and the *bp* parameter addresses the *buf* structure containing the request. The queue of requests is sorted in ascending order by the *disksort* routine to optimize disk head movement.

Parameters

The *xxtab* parameter is the address of an *iobuf* data structure declared within the driver to form the head of the I/O request queue.

The *bp* argument is a *buf* * data structure that points to the I/O request to be added to the queue.

Notes

This routine is only for use with block device drivers.

dma_alloc

allocates a DMA channel

Syntax

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

int
dma_alloc(chan, mod)
unsigned chan, mod;
```

Description

The *dma_alloc* routine allows dynamic allocation of a DMA channel.

Parameters

The *chan* argument specifies the channel to be allocated. Possible values are:

8-Bit Channels:	DMA_CH0, DMA_CH1, DMA_CH2, DMA_CH3
16-Bit Channels:	DMA_CH5, DMA_CH6, DMA_CH7

Channel 4 is not available. Other channels may be permanently allocated by system drivers. Consult the **/usr/adm/messages** file for which channels are in use. Use *printcfg*(K) in your driver initialization routine to display the DMA channel that you select.

The *mod* argument can have one of two values:

DMA_BLOCK	wait until the channel is available. If used, do not call from an interrupt routine or an *xxinit* routine.
DMA_NBLOCK	return immediately with a return status of 0 (zero) if the channel was not free at this time.

If *mod* specifies blocking, the *dma_alloc* routine does not return until the requested channel is available. It sleeps until the channel is released and always returns non-zero. If *mod* specifies non-blocking, the *dma_alloc* routine immediately returns non-zero if the channel is available, and zero if it is not. The blocking option cannot be used at interrupt time, but the non-blocking option can be.

Make certain that your DMA channel has been allocated before beginning your operations.

Example

An example of how to use this routine is:

```
#include "sys/errno.h"
#include "sys/dma.h"

                extern struct dmareq dma_request;

        /* Allocate channel 1.  If not */
        /* currently available, wait.  */

        if ( dma_alloc( DMA_CH1, DMA_BLOCK ) == 0 )
        {
                seterror( EIO );
                return;
        }

        /* If channel is successfully allocated, */
        /* then begin DMA streaming */

        dma_start( &dma_request );
```

See Also

dma_param(K), dma_start(K), dma_relse(K), dma_enable(K), dma_resid(K), printcfg(K)

dma_breakup

sizes DMA request into 512-byte blocks

Syntax

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

Description

The *dma_breakup* routine breaks up Direct Memory Access (DMA) I/O requests into 512-byte units of contiguous memory to avoid limitations imposed by DMA controllers. *dma_breakup* is called by the *physio*(K) routine, indirectly. Place the *dma_breakup* call in a subroutine and then call the name of that routine from *physio*.

dma_breakup first determines the correct block number of the data being passed.

If a read is being requested, *xxstrategy* is called to get a buffer header. If a buffer header is not available from the call to *xxstrategy*, *dma_breakup* goes to sleep until one is free. While sleeping, the request for a buffer is protected from signals, and from interrupts occurring at or below *spl6*(K). When a buffer is free, data is read from user space.

A write request is similar, except that the data is copied to a kernel page from user space before *xxstrategy* is called. Again, *sleep* is called to wait for a free buffer header.

After ensuring that a buffer header is free, DMA transfer starts. If an error is caused by reaching the end of the media, ENXIO is returned. During DMA transfer, *xxstrategy* is called to put the current buffer on the buffering mechanism. Each time *xxstrategy* is called, *sleep* is also called to wait until buffering occurs.

NOTE: Use *splx*(K) to save your *spl* setting before calling *dma_breakup* because *dma_breakup* calls *spl0*(K) and cancels all previously set *spl* levels.

dma_breakup depends on the following fields of the **user** structure that are set up by the kernel when the I/O request is passed to the driver:

- **u.u_base** — the virtual base address for the calling program in user space
- **u.u_count** — the number of bytes to be transferred
- **u.u_offset** — offset into the file from/to which data is transferred.

In addition, the driver should set **b_flags** to indicate the type of transfer. Possible values are B_READ or B_WRITE. *dma_breakup* calls *sleep*(K) and therefore can only be called from a non-interrupt routine.

Parameters

The parameters to *dma_breakup* are as follows:

xxstrategy	The name of the *xxstrategy* driver routine.
bp	A pointer to the *buf* structure.

Return Value

None. However, the following values may change:

- **b_flags** — if insufficient memory is available for allocation, **b_flags** is ORed with B_ERROR and B_DONE, and **b_error** is set to EAGAIN.
- **b_un.b_addr** — set to the virtual kernel address.
- **b_blkno** — changed.
- **b_bcount** — changed.
- **u.u_segflg** — set to zero.
- **u.u_base, u.u_count, u.u_offset** — **u.u_base** and **u.u_offset** are incremented by the number of characters to be transferred; **u.u_count** is decremented.

dma_enable

begins DMA transfer

Syntax

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

int
dma_enable(chan)
unsigned chan;
```

Description

This routine starts a DMA transfer. There is no return value. This routine clears the mask register on the controller to let a DMA transfer begin.

Parameters

The *chan* argument specifies the DMA channel to be used. Possible values are:

8-Bit Channels:	DMA_CH0, DMA_CH1, DMA_CH2, DMA_CH3
16-Bit Channels:	DMA_CH5, DMA_CH6, DMA_CH7

Channel 4 is not available. Other channels may be permanently allocated by system drivers. Consult the **/usr/adm/messages** file for which channels are in use. Use *printcfg* in your driver initialization routine to display the DMA channel that you select.

See Also

dma_param(K), dma_start(K), dma_relse(K), dma_alloc(K), dma_resid(K), printcfg(K)

dma_param

sets up a DMA controller chip for DMA transfer

Syntax

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

int
dma_param(chan, mode, addr, cnt)
unsigned chan, mode;
paddr_t addr;
long cnt;
```

Description

This routine sets up the controller chip for a DMA transfer. The *dma_param* routine masks the DMA request line on the DMA controller. Only call *dma_param* after a DMA channel has been allocated for the driver by *dma_alloc*(K) or *dma_start*(K). In the case of a driver using *dma_start*, *dma_param* is called by the routine pointed to by the *d_proc* member of the *dmareq* structure. The *dma_param* routine has no return value.

Parameters

The *chan* argument specifies the DMA channel to be used. Possible values are:

8-Bit Channels:	DMA_CH0, DMA_CH1, DMA_CH2, DMA_CH3
16-Bit Channels:	DMA_CH5, DMA_CH6, DMA_CH7

Channel 4 is not available. Other channels may be permanently allocated by system drivers. Consult the **/usr/adm/messages** file for which channels are in use. Use *printcfg* in your driver initialization routine to display the DMA channel that you select.

The *mode* argument specifies whether this is a read or write transfer. The options are:

- DMA_Rdmode (0x44). This option specifies a transfer from a device to memory.
- DMA_Wrmode (0x48). This option specifies a transfer from memory to a device.

The *addr* argument specifies the address where the data is copied from or to.

The *cnt* argument specifies the number of bytes or words to transfer.

Example

For example, the function mentioned in the example for *dma_start foo_proc*, might contain this code:

```
foo_proc( dp )
struct dmareq *dp;
{
        .
        .
        dma_param(dp->d_chan, dp->d_mode,
                dp->d_addr, dp->d_cnt);
        dma_enable( dp->d_chan );
        .
        .
}
```

See Also

dma_enable(K), dma_start(K), dma_relse(K), dma_alloc(K), dma_resid(K), printcfg(K)

dma_relse

releases previously allocated DMA channel

Syntax

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

int
dma_relse(chan)
unsigned chan;
```

Description

The *dma_relse* routine releases a DMA channel previously allocated with *dma_alloc*(K) or *dma_start*(K). This routine should be called during the interrupt signaling completion of the DMA transfer or as soon as completion is detected (if polling is being used). This routine has no return value. If you intend to share DMA channels, you should use this routine. Sharing DMA channels is highly recommended.

If no *dmareq* structures are in the pending-request queue, *dma_relse* releases the channel, wakes up any processes sleeping on the channel, and exits. Otherwise it performs the next request on the queue by calling the *xxd_proc* routine with a pointer to the *dmareq* structure as a parameter. Because *xxd_proc* may be called during another driver's interrupt, the *xxd_proc* routine should be as minimal as possible to accomplish its task.

Parameters

The argument *chan* is the DMA channel to be released. Possible values are:

8-Bit Channels:	DMA_CH0, DMA_CH1, DMA_CH2, DMA_CH3
16-Bit Channels:	DMA_CH5, DMA_CH6, DMA_CH7

Channel 4 is not available. Other channels may be permanently allocated by system drivers. Consult the **/usr/adm/messages** file for which channels are in use. Use *printcfg* in your driver initialization routine to display the DMA channel that you select.

Example

To release the channel that was allocated in the previous allocation examples:

```
/* finished with DMA for now, release channel */
                dma_relse( DMA_CH1 );
```

See Also

dma_enable(K), dma_start(K), dma_param(K), dma_alloc(K), dma_resid(K), printcfg(K)

dma_resid

returns the number of bytes not transferred during a DMA request

Syntax

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

long
dma_resid(chan)
unsigned chan;
```

Description

This routine returns the number of bytes not transferred by the DMA request.

Parameters

The *chan* argument specifies the DMA channel to be queried. Possible values are:

8-Bit Channels:	DMA_CH0, DMA_CH1, DMA_CH2, DMA_CH3
16-Bit Channels:	DMA_CH5, DMA_CH6, DMA_CH7

Channel 4 is not available. Other channels may be permanently allocated by system drivers. Consult the **/usr/adm/messages** file for which channels are in use. Use *printcfg* in your driver initialization routine to display the DMA channel that you select.

Return

The *dma_resid* routine returns the number of bytes that were not transferred.

See Also

dma_enable(K), dma_start(K), dma_param(K), dma_alloc(K), dma_relse(K), printcfg(K)

dma_start

queues DMA request

Syntax

#include "sys/dma.h"

int
dma_start(arg)
struct dmareq *arg;

Description

The *dma_start* routine queues a DMA request for later execution when the requested channel is available. *dma_start* can be used in initialization or interrupt routines. The format of the *dmareq* structure is as follows:

```
struct dmareq {
struct dmareq    *d_nxt;       /* reserved */
unsigned short   d_chan;       /* specifies channel */
unsigned short   d_mode;       /* direction of transfer */
paddr_t          d_addr;       /* physical src or dst */
long             d_cnt;        /* number of bytes or words */
int              (*d_proc)();  /* address of routine to call */
char             *d_params;    /* pointer to params for d_proc */
};
```

The *dmareq* structure contains enough information to specify the transfer, the address of a routine to call when the channel is available, and an address of further data that may be needed by the *xxd_proc* routine (the **d_proc** field of the structure).

Possible values for **d_chan** are:

8-Bit Channels:	DMA_CH0, DMA_CH1, DMA_CH2, DMA_CH3
16-Bit Channels:	DMA_CH5, DMA_CH6, DMA_CH7

Channel 4 is not available. Other channels may be permanently allocated by system drivers. Consult the **/usr/adm/messages** file for which channels are in use. Use *printcfg* in your driver initialization routine to display the DMA channel that you select.

dma_start sets up the kernel to allocate the DMA channel for the driver. By filling in the **d_chan** field with the channel you want, the **d_mode** with the mode you want, and the *xxd_proc* with a pointer to the routine to be called once the DMA channel is allocated, the driver can request that the kernel allocate a channel.

When the channel is allocated, the routine pointed to by *xxd_proc* is called with a pointer to the *dmareq* structure. At this point, the DMA channel has been allocated as if the driver had done so with *dma_alloc*.

If the routine was not able to allocate the channel immediately, but had to queue your request, this routine will return a 0.

Example

For example, to allocate DMA channel 1 for reading and with *foo_proc* as the *xxd_proc* routine, use:

```
/* set up dma structure */
extern int foo_proc();

struct dmareq foo_req = {      /* DMA request structure: */
        (struct dmareq *)0,    /*          d_nxt         */
        DMA_CH1,               /*          d_chan        */
        DMA_Rdmode,            /*          d_mode        */
        (paddr_t)0,            /*          d_addr        */
        (long)0,               /*          d_cnt         */
        foo_proc,              /*          d_proc        */
        (char *)0,             /*          d_params      */
};
    . . .

        dma_start( &foo_req );
        /* we don't care if we are queued or not */
        return;
```

Parameters

The *arg* argument is a pointer to the *dmareq* structure that specifies the transfer that is required.

Return

If the channel is available, it is marked as "busy," and *arg->d_proc* is called at *spl6*(K) with a pointer to *arg* as a parameter. The *dma_start* routine then returns a non-zero value.

If the channel is not available, the structure **arg* is linked to the end of a list of pending requests, and *dma_alloc*(K) simply returns 0.

Note

The kernel routines contained in the *xxd_proc* routine are executed at *spl6*(K) and should observe all the normal rules of an interrupt routine. Specifically, this means that no assumptions about the currently running process may be made. In addition, the interrupt priority level should not be lowered, and *sleep*(K), *delay*(K), or other routines that call *sleep* cannot be used.

See Also

dma_enable(K), dma_resid(K), dma_param(K), dma_alloc(K), dma_relse(K)

emdupmap

duplicates channel mapping

Syntax

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

int
emdupmap(tp, ntp)
struct tty *tp, *ntp;
```

Description

The *emdupmap* routine duplicates the mapping of a given channel for a new channel.

Parameters

The *tp* parameter is a pointer to the *tty* structure for the line the mapping should be duplicated from.

The *ntp* parameter is a pointer to the *tty* structure for the line where the characters are to be placed.

Return Value

This routine has no return value, but the routine can return immediately with no work done if both arguments point to the same *tty* structure.

Notes

Note that this routine is for use only within character device drivers.

See Also

emunmap(K)

emunmap

disables mapping on a channel

Syntax

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

int
emunmap(tp)
struct tty *tp;
```

Description

The *emunmap* routine disables mapping on a channel.

Parameters

The *tp* parameter is a pointer to the *tty* structure of the mapped line that is to have the mapping disabled.

Return Value

This routine has no return value.

Notes

Note that this routine can be used only with character device drivers.

See Also

emdupmap(K)

flushtlb

flushes the translation lookaside buffer

Syntax

void
flushtlb( )

Description

When accessing an I/O port above 0x1000, a driver must flush the translate lookaside buffer (TLB) to prevent corruption of the I/O address. In addition, the call to *flushtlb* must be protected by a call to *spl7* to prevent interrupts from occurring while *flushtlb* is operating.

Example

The following example is for a driver routine that gets a byte from an I/O port and checks the address before beginning processing. A driver should also have a similar routine for writing data to an I/O port.

```
int
getbyte(port)
{
        int x, ret;
        /*
         * If port address is less than hex 1000,
         *   get byte and return.
         */
        if (port < 0x1000)
                return( inb(port) );
                            /*          else,       */
        x = spl7();         /* block all interrupts */
        flushtlb();         /* flush TLB            */
        ret = inb(port);    /* get byte from port   */
        splx(x);            /* reset previous spl   */
        return(ret);        /* return byte to caller */
}
```

See Also

inb(K), spl(K)

fubyte

gets a character from user data space

Syntax

```
int
fubyte(src)
unsigned char *src;
```

Description

The *fubyte* routine retrieves (fetches) one character from the user's data space. If you are fetching data from **u.u_base**, consider using *cpass*(K) in that **u.u_count**, **u.u_offset**, and **u.u_base** are updated for you, and error handling is provided via **u.u_error**. If any chance exists that a -1 may be contained in the data that you are receiving from user space, use *copyin*(K) instead of *fubyte* (or *cpass*). A -1 in the data is usually associated as an error condition.

This routine must not be called from an interrupt or *xxinit* routine.

Parameters

The argument *src* points to the address from which the byte is to be copied.

Return Value

The value of the retrieved byte is returned. If -1 is received, then an error occurred and **u.u_error** should be set to EFAULT.

See Also

fuword(K), cpass(K)

fuword

gets one 32-bit word from user data space

Syntax

int
fuword(src)
unsigned int *src;

Description

The *fuword* routine retrieves (fetches) one 32-bit word from the user's data space. This routine must not be called from an interrupt or an *xxinit* routine.

Parameters

The argument *src* is an address in user space from which the word is copied from.

Return Value

The value of the retrieved 32-bit word is returned. If an error occurs, -1 is returned and you should set **u.u_error** to EFAULT.

See Also

fubyte(K)

getc, getcb, getcbp, getcf

read clist buffers

Syntax

```
int
getc(cp)
struct clist *cp;

struct cblock*
getcb(cp)
struct clist *cp;

int
getcbp(p, cp, n)
struct clist *p;
char *cp;
int n;

struct cblock *
getcf( )
```

Description

The *getc* routine moves one character from the *clist* buffer for each call.

The *getcb* routine moves one *cblock* from the *clist* buffer for each call.

The *getcbp* routine copies characters from the specified *clist*, *p*, to the buffer addressed by the *cp* argument.

The *getcf* routine takes a *cblock* from the freelist and returns a pointer to it.

Parameters

cp specifies the *clist* buffer from which characters are moved by *getc*.

The pointer *cp* specifies the *clist* buffer the *cblocks* are moved from by *getcb*.

The pointer *p* specifies the *clist* buffer the characters are copied from by *getcbp*. The argument *cp* is a *char* * that addresses the buffer the characters are copied to.

The value of *n* is the number of characters to be copied (which should denote the maximum size of the available buffer).

Return Value

The *getc* routine returns the next character in the buffer or -1 if the buffer is empty.

The *getcb* routine returns a pointer to the first *cblock* on the *clist* or NULL if the *clist* is empty.

The *getcbp* routine returns the number of characters actually copied (which is less than or equal to *n*).

getcf returns a pointer to a *cblock* if available. Otherwise, the routine returns NULL.

Notes

Note that these routines can be used only within character device drivers.

See Also

putc(K)

getchar

gets one character of input

Syntax

int
getchar( )

Description

getchar can be used to temporarily halt execution of the kernel, and get input from a user.

Return Value

getchar returns the character typed at the keyboard.

Example:

```
        debug = getchar();
        debug -= '0';
```

See Also

putchar(K)

geteblk, getablk

gets a buffer from the block buffer pool

Syntax

struct buf *
getablk( )

struct buf *
geteblk( )

Description

The *geteblk* routine acquires a free buffer from the block buffer pool. The pointer returned by this routine addresses a buffer that can be used as required. The buffer can subsequently be returned to the buffer pool by calling *brelse*(K) or *iodone*(K).

getablk calls *geteblk* directly. *getablk* is provided for compatibility only and may go away in future releases.

OR the **b_flags** field of the **buf** structure with B_WANTED if *geteblk* should sleep if a buffer is not available. When sleeping is requested, it is performed below PZERO and is not affected by signals.

When a buffer is allocated, *geteblk* ORs **b_flags** with B_BUSY and B_AGE, **b_back** and **b_forw** are set to the same buffer pointer that is returned, **b_dev** is set to NODEV, and **b_bcount** is set to SBUFSIZE. SBUFSIZE is defined in **sys/fs/s5param.h** and varies in size according to the file system size.

Return Value

This routine returns a *struct buf* * that addresses the allocated buffer.

Notes

This routine may be used only by block device drivers. This routine calls *spl0*(K) to enable all interrupts. This may change previously set interrupt levels.

See Also

brelse(K), iodone(K)

inb, outb

reads a byte from or writes a byte to an I/O address

Syntax

```
int
inb(read_addr)
int read_addr;

int
outb(write_addr, value)
int write_addr;
char value;
```

Description

The *inb* routine reads a byte from the I/O address specified by the parameter *read_addr*.

The *outb* routine writes the byte specified by *value* to the physical I/O address specified by *write_addr*.

Warning

If the specified read or write address is above 0x1000, call *flushtlb*(K) before calling *inb* or *outb*. Refer to the *flushtlb* manual page for more information.

Parameters

The value of *read_addr* is an integer specifying the physical I/O address from which to read.

write_addr is an integer specifying the physical I/O address to which to write.

value is a byte to write to *write_addr*.

Return

inb returns a byte. physical I/O address specified by *read_addr* is returned. *inb* returns an integer whose high byte or bytes have been cleared. Only the low byte is meaningful.

Example

To read a byte register at I/O address 0x300 you could use the following lines of code:

```
char val;
val = (char) inb(0x300);
```

To write the 8-bit value 0xF to a byte register at I/O address 0x300 you could use the following line of code:

```
outb(0x300,  0xF);
```

See Also

flushtlb(K), ind(K), inw(K), repinsb(K), copyin(K)

ind, outd

reads, writes a 32-bit word to a physical I/O address

Syntax

```
int
ind(read_addr)
int read_addr;

int
outd (write_addr, value)
int write_addr, value;
```

Description

The *ind* function reads a 32-bit word from the physical I/O address specified by *read_addr*.

outd writes a 32-bit value to the physical I/O address specified by *write_addr*.

Warning

If the specified read or write address is above 0x1000, call *flushtlb*(K) before calling *ind* or *outd*. Refer to the *flushtlb* manual page for more information.

Parameters

The value of *read_addr* is an integer that specifies the physical I/O address to be read from.

write_addr is the physical I/O address being written to. *value* is the 32-bit word being written to *write_addr*.

Return

ind returns the 32-bit value read from the I/O address *read_addr*. *outd* has no return value.

Example

To read a 32-bit value from I/O address 0x300 you could use the following code:

```
int val;
val = ind(0x300);
```

To write the 32-bit value 0xFFFF00 to I/O address 0x300 you could use the following code:

```
outd(0x300, 0xFFFF00);
```

See Also

flushtlb(K), inb(K), inw(K), repinsb(K), copyin(K)

inw, outw

reads, writes a 16-bit word from or to a physical I/O address

Syntax

```
int
inw(read_addr)
int read_addr;

int
outw (write_addr, value)
int write_addr, value;
```

Description

The *inw* function reads a 16-bit word from the physical I/O address specified by *read_addr*. *outw* writes a 16-bit word to the physical I/O address specified by *write_addr*.

Warning

If the specified read or write address is above 0x1000, call *flushtlb*(K) before calling *inw* or *outw*. Refer to the *flushtlb* manual page for more information.

Parameters

read_addr is an integer that specifies the physical I/O address to be read from.

write_addr is the physical I/O address being written to. *value* is the 16-bit word bring written to *write_addr*.

Return

A 32-bit integer whose high 2 bytes are set to zero is returned by *inw*. *outw* has no return value.

Example

To read a 16-bit register at I/O address 0x300 you could use the following lines of code:

```
short int val;
val = (short int) inw(0x300);
```

To write the 16-bit value 0xFF0 to I/O address 0x300 you could use the following code:

```
outw(0x300, 0xFF0);
```

See Also

flushtlb(K), inb(K), ind(K), repinsb(K), copyin(K)

iodone

signals I/O completion

Syntax

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

Description

The *iodone* routine completes a block driver's I/O request and wakes up all processes waiting completion of block I/O requests. This routine is generally placed in a driver's interrupt routine. *iodone* calls *brelse*(K) to release the buffer, followed by *wakeup* to awaken the sleeping processes. *iodone* ORs B_DONE into *b_flags*.

Parameters

The *bp* argument specifies a *struct buf* * that addresses the buffer.

Notes

Note that this routine can be used only with block device drivers.

See Also

brelse(K), iowait(K)

iowait

wait for I/O completion

Syntax

int
iowait(bp)
struct buf * bp;

Description

The *iowait* routine is called by the higher levels of the kernel I/O system to wait for the completion of an I/O operation specified by the buffer addressed by the *bp* parameter. This routine should not be called within an interrupt routine since it may call the *sleep*(K) routine.

iowait transfers any errors found in **bp->b_error** to **u.u_error** for the process indicated in the *buf* header.

Parameters

The *bp* argument specifies a *struct buf* * that addresses the buffer involved in the I/O operation.

Return Value

There is no result returned. The calling process continues when the I/O operation is complete.

Notes

Note that this routine may only be used with block device drivers.

See Also

iodone(K), brelse(K)

longjmp

ends current system call with error

Syntax

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

int
longjmp(u.u_qsav)
label_t u.u_qsav;
```

Description

The *longjmp* routine passes control of the current process to the end of the current system call. The system call then returns to user space with the external variable *errno* set to the EINTR error code. Each time a driver is called by the kernel, registers are saved in **u.u_qsav**. The *longjmp* routine restores these registers, sets **u.u_error** to EINTR, and then jumps to the end of the current system call. The effect from a driver's perspective is that the jump returned control of the process back to the calling user process with an error set.

This routine must never be called from an initialization or interrupt routine.

longjmp is called by *sleep*(K) if a signal is received directed at the sleeping process, and if the priority argument to *sleep* is not ORed with PCATCH.

longjmp(K) differs from the user space *longjmp*(S) in three ways: the intent is different, the arguments are different, and kernel *longjmp* is **not** used in conjunction with a *setjmp* routine. Use of the *setjmp* kernel routine is not supported.

Parameters

u.u_qsav contains register information. This is the only argument that should ever be passed to *longjmp*.

See Also

sleep(K), setjmp(S)

major, makedev, minor

returns major, new device number, or minor device number

Syntax

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

int
major(device-number)
dev_t device-number;

dev_t
makedev(major-num, minor-num)
int major-num, minor-num;

int
minor(device-number)
dev_t device-number;
```

Description

The *major* macro returns the major device number from a device number. *minor* returns the minor device number.

The *makedev* macro returns a new device number from major and minor device numbers.

Parameters

device-number is a *short* integer device number that contains both the major and minor device numbers.

memget

allocates contiguous memory at initialization

Syntax

```
int
memget(pages)
int pages;
```

Description

The *memget* routine is used to obtain permanent, contiguous memory for the driver at initialization time. It is intended for memory that the driver will always have and use. Its argument is the size of memory in pages. Use the macro *btoc*(K) to calculate the number of pages from the number of bytes required. *memget*'s return value is also in pages, so the *ctob*(K) macro must be used to translate the return value of *memget* into a kernel virtual address. Both *ctob* and *btoc* are defined in the file **/sys/sysmacros.h**.

Parameters

pages is the number of pages to allocate.

Warning

This routine is intended for use in a driver's initialization routine for use before any user processes have been run. Calling *memget* in other routines can result in a caller sleeping forever. If physically contiguous memory is not immediately available, *memget* goes to sleep with periodic checks, but never rearranges pages to obtain the memory. Thus if the memory is not available during a check, *memget* returns to sleep, and may never find available memory.

Return Value

The page frame number of the first frame of memory allocated is returned.

Example

To obtain a permanent 4K buffer for a driver, use the following code statement:

```
char *always;

always = (char *) ctob( memget( btoc( 0x1000 ) ) );
```

paddr

returns virtual address pointer to block data

Syntax

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

paddr_t
paddr(bp)
```

Description

This macro returns a virtual address pointer to the data contained in a block driver buffer header. *paddr* provides a pointer to the **b_un.b_addr** member of the *buf* structure. Use this macro to ensure portability between releases of System V.

Note

This macro returns a **virtual** address, not a physical address. This may be misleading since the name of the macro implies a physical address.

Parameters

bp	Pointer to a buffer header

Return Value

A virtual address pointing to the **b_un.b_addr** field of the *buf* structure.

panic

halts the system

Syntax

void
panic(string)
char * string;

Description

The *panic* routine takes a parameter *string* that points to a string and prints the string on the system console and halts the system. It is called whenever an unrecoverable kernel error is encountered. This routine should be called only under extreme circumstances.

Parameters

The variable *string* is an address of a string that describes the reason for the system failure.

Example

```
panic("the cpu has melted down");
```

physio, physck

raw I/O for block drivers

Syntax

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

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

Description

The *physck* routine ensures that a requested raw I/O request can be serviced by the device being read from or written to. The *nblocks* argument to *physck* is the maximum number of 512-byte blocks on the device, or in the disk partition. The number of blocks is converted to bytes to find the size limit and then compared with **u.u_offset**. If **u.u_offset** is greater than the size limit, and *rwflag* is B_WRITE, then **u.u_error** is set to ENXIO and a 0 (zero) is returned. If **u.u_offset** is greater than or equal to the limit, and *rwflag* is B_READ, then 0 (zero) is returned and no error is set. If **u.u_offset** plus **u.u_count** is greater than the limit, then **u.u_count** is reduced by the number of bytes that it differs from the limit and the test is completed. *physio* returns 1 on all successful tests.

The *physio* routine provides a raw (direct) I/O interface for block-device drivers. It validates the request, builds a buffer header, locks the process in core, and calls a *routine*, to queue the request. The *routine* usually called is one containing a call to *dma_breakup*(K), or *routine* may be a call to an *xxstrategy* routine. Refer to Appendix B, "Sample Block Driver" for an example of a *physio* call in an *xxread* routine where a *dma_breakup* routine is called.

Warning

Before calling *physio*, make sure that the buffer is not busy (**b_flags** contains B_BUSY).

Notes

If the data transfer crosses a 64K segment boundary, *physio* may break the request into 3 pieces. If the data request crosses a 4K page boundary, the request is broken into BSIZE pieces. BSIZE is defined in **sys/fs/s5param.h** and varies by file system size.

Parameters

nblocks	the number of 512-byte blocks to be read from the device or written to the device.
rwflag	an I/O flag. Set to B_READ for reading; set to B_WRITE for writing.
routine	the address of a routine to be executed, generally *xxbreakup* or *xxstrategy*.
bp	a pointer to the buffer header describing the request to be filled. Ensure that the buffer is not busy before calling *physio*. Set the *bp* argument to NULL to have *physio* allocate a buffer. The following example illustrates such a call:

```
physio(xxbreakup, (struct buf *) NULL,
       dev, B_READ);
```

dev	an integer specifying the device number (includes both the major and minor device numbers)
rwflag	an I/O flag. Set to B_READ for reading; set to B_WRITE for writing. The possible values are: B_READ — read from disk to user memory, and B_WRITE — write from user memory to disk.

Note

The **u.u_base , u.u_count ,** and **u.u_offset** values must be set up prior to the *physio* call, and must point to the appropriate user-data area.

See Also

db_read(K), db_write(K), dma_breakup(K), pio_breakup(K)

pio_breakup

breaks up programmed I/O requests

Syntax

```
int
pio_breakup(xxstrategy, bp, maxsecsz)
int (*xxstrategy)( );
struct buf *bp;
int maxsecsz;
```

Description

This routine breaks up programmed I/O requests into *maxsecsz* pieces, and either reads data from the user process or writes data to the user process depending on how **b_flags** (pointed to by *bp*) is set. The address in the user process from which data is read from or written to is pointed to by the **u.u_base** field in the **user** structure. *pio_breakup* is used to break up requests across page boundaries.

A buffer header is allocated and filled using information from the *bp* **buf** pointer. Then the *xxstrategy* routine is called. *sleep*(K) is called to wait until *xxstrategy* completes.

If an error occurs, the error code is passed from **b_error** to **u.u_error**, the allocated buffer is released, and *pio_breakup* returns. In addition, the following fields are set in the buffer header pointed to by *bp*:

- *b_resid* — Set to the original value of **b_count** from the passed in buffer header
- **b_flags** — ORed with B_DONE and B_ERROR
- **b_error** — Set to the value of **u.u_error**

Notes

pio_breakup calls *spl0*(K) which may alter previously set *spl* levels in your driver.

This routine adjusts **u.u_base, u.u_offset,** and **u.u_count**

pio_breakup must not be called from an interrupt or *xxinit* routine.

Parameters

xxstrategy	The name of the *xxstrategy* routine for a driver
bp	Buffer header pointer
maxsecsz	The maximum number of blocks to move in each transfer. Typically, this value is 256, but it varies by device.

Return Value

None.

See Also

dma_breakup(K)

printcfg

displays driver initialization message

Syntax

int
printcfg(name, base, offset, vec, dma, fmt, args)
char *name, *fmt;
unsigned base, offset;
int vec, dma;
void args;

Description

This routine displays configuration information for a device on the console. The information is also stored in **/usr/adm/messages** by a background error logging program. Use this routine when displaying messages in the *xxinit* routine of a driver to maintain consistency with other drivers' initialization messages.

The following is displayed by *printcfg*:

```
device    address          vector    dma    comment
%name     0xbase-0xend     vec       dma    fmt
```

Where:

name	Driver name
base	The base of I/O addresses
end	The summed value of *base* + *offset*.
vec	Interrupt vector (in octal)
dma	Direct memory address (DMA) channel (in decimal)

Parameters

name	A character string containing the driver name. This string is required.
base	A hexadecimal address indicating the base of I/O addresses. Set to 0 (zero) to omit.

offset	A hexadecimal value indicating the range of I/O addresses from base. Set to 0 (zero) to omit.
vec	A two digit octal interrupt vector number derived from switchable settings on the card. Set to -1 to omit.
dma	A decimal value indicating the direct memory access (DMA) channel. Set to -1 to omit.
fmt	A string to be displayed similar to the format argument to *printf*(S). *fmt* also accepts specifications for displaying the *arg* variables. The supported specifications are:

Type	**Description**
%b	two-digit hexadecimal byte
%c	character
%d	signed decimal
%o	unsigned octal
%s	string (character pointer)
%x	hexadecimal (prints leading zeros)

The specification values can be indicated in either upper or lower case. Field length specifications cannot be used for arguments. For example, *%9d* is not permitted. Escaped characters such as *n* (new line), *t* (tab), *r* (return), and so on are C language features supported by the C compiler and thus are supported in this kernel routine. A *n* character is automatically appended to the end of *fmt*.

args	Optional variables to be displayed using *fmt*.

This routine is not interrupt-driven and therefore suspends all other system activities while executing.

Leading zeros are not displayed for the *base* value, but are displayed if the *%x* or *%b* specifications are used in the *fmt* argument.

This routine does not function properly on consoles running *layers*(C).

See Also

cmn_err(K), printf(K), printf(S), layers(C)

printf

print a message on the console

Syntax

int
printf(format, arg1, arg2, ...)
char *format;
void arg1, arg2;

Description

The kernel *printf* routine prints error messages and debugging information on the system console. In addition, all messages are stored in **/usr/adm/messages** by a background program error handler. *printf* is a simplified version of the standard C library *printf*(S) routine.

Parameters

The parameters are:

format A string to be displayed similar to the format argument to *printf*(S). *format* also accepts specifications for displaying the *arg* variables. The supported specifications are:

Type	**Description**
%b	two-digit hexadecimal byte
%c	character
%d	signed decimal
%o	unsigned octal
%s	string (character pointer)
%x	hexadecimal (prints leading zeros)

Specification values can be indicated in either upper or lower case. Field length specifications cannot be used for arguments. For example, *%9d* is not permitted. Escaped characters such as \n (new line), \t (tab), \r (return), and so on are C language features supported by the C compiler and thus are supported in this kernel routine.

arg1, *arg2* Optional variables to be displayed using the *format*.

Notes

This routine is not interrupt-driven and therefore suspends all other system activities while executing.

This routine is similar to standard C library function *printf*(S), except that only the formats specified here are valid, and precision is not supported.

This routine does not function properly on consoles running *layers*(C). As it is impossible for a driver to know if a console is running *layers*, use of *printf*(K) for other than debug purposes is not recommended. Use *cmn_err*(K) instead.

Example

It is often useful to use *printf* for driver debug statements. For example in your *xxioctl* routine you might do this:

```
xxioctl(dev, cmd, arg, mode)
dev_t dev;
int cmd, arg, mode;
{
        printf("dev= %x, cmd= %d, addr of arg= %x, mode= %x\n",
                dev, cmd, arg, mode);
}
```

See Also

cmn_err(K), printf(S)

psignal

sends signal to a process

Syntax

```
#include "sys/proc.h"
#include "sys/signal.h"

int
psignal(p, sig)
proc_t *p;
int sig;
```

Description

The *psignal* routine sends the specified signal *sig* to the process specified by *p*. **proc_t** is another name for the **proc** structure and is defined in sys/**proc.h**.

Parameters

p is a pointer to the process to which the signal is sent. At task time it is **u.u_procp** (see **sys/user.h**). If you want to be able to kill a process at interrupt time you need to store **u.u_procp** in a global variable.

sig is the number of the signal to be sent. For more information about possible signals, see the /**sys/signal.h** header file.

See Also

signal(K)

ptok, ktop

converts virtual and physical addresses

Syntax

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

int
ptok(x)
int x;

int
ktop(x)
int x;
```

Description

The *ptok* macro converts a physical address *x* to a kernel virtual address.

The *ktop* macro converts kernel virtual address *x* to a physical address.

Parameters

x is the address to be switched.

Return Value

The new kernel virtual address is returned by *ptok*. The new physical address is returned by *ktop*.

putc, putcb, putcbp, putcf

write to clists

Syntax

```
int
putc(c, cp)
int c;
struct clist *cp;

int
putcb(cbp, cp)
struct cblock *cbp;
struct clist *cp;

int
putcbp(p, cp, n)
struct clist *p;
char *cp;
int n;

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

Description

The *putc* routine moves one character to the *clist* buffer for each call. *putc* contains a "critical" section of code and should be protected from interrupts when called.

The *putcb* routine moves one *cblock* to the *clist* buffer for each call.

The *putcbp* routine appends characters from a buffer to the *clist* given as an argument. If the current *cblock* has no room for the requested characters, *putcbp* gets new *cblock*s from the free list as needed.

The *putcf* routine puts the specified *cblock* onto the freelist.

Parameters

The value of *c* is an integer that specifies the character to be moved.

A pointer *cp* specifies the *clist* buffer where the character is placed.

The pointer *cbp* specifies the *cblock* to be moved by *putcb*.

The pointer *cp* specifies the *clist* buffer to where the *cblock* is moved by *putcb*.

The argument *cbp* specifies a pointer to a *cblock*.

Return Value

The *putc* routine returns 0 if it places the specified character in the buffer, or returns -1 if there is no free space. The *putcb* routine returns 0 after placing the specified *cblock* in the buffer.

Notes

These routines can be used only with character device drivers.

See Also

getc(K)

putchar

prints a character on the console

Syntax

```
int
putchar(c)
char c;
```

Description

The *putchar* routine is used by the *printf*(K) and *panic*(K) routines. This routine puts one character on the console, doing a ''busy wait'' rather than depending on interrupts. This means that all other system activities are suspended while *putchar* is executing.

Parameters

c is the character that is printed on the console.

See Also

getchar(K)

repins: repinsb, repinsw, repinsd, repoutsb, repoutsw, repoutsd

reads and writes streams of device data

Syntax

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

int
repinsb(dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;

int
repinsw(dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;

int
repinsd(dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;

int
repoutsb (dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;

int
repoutsw(dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;

int
repoutsd(dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;
```

Description

The *repins* routines are used to read streams of data from an I/O port on a device to an array of size *cnt* whose base begins at the kernel virtual address specified by *kv_addr*. These routines are typically used for reading from disk and SCSI devices. These routines assume that the virtual address specified is a valid kernel address currently in RAM.

repinsb reads a stream of bytes from an I/O address to a kernel virtual address.

repinsw reads a stream of 16-bit words from an I/O address to a kernel virtual address.

repinsd reads a stream of 32-bit words from an I/O address to a kernel virtual address.

The *repout* routines are used to write streams of data to an I/O port on a device from an array of size *cnt* whose base begins at the kernel virtual address specified by *kv_addr*. These routines are typically used for writing to disk and SCSI devices. These routines assume that the virtual address specified is a valid kernel address currently in RAM.

repoutsb writes a stream of bytes to an I/O address from a kernel virtual address.

repoutsw writes a stream of 16-bit words to an I/O address from a kernel virtual address.

repoutsd writes a stream of 32-bit words to an I/O address from a kernel virtual address.

Warning

If the specified read or write I/O port address is above 0x1000, call *flushtlb*(K) before calling a *repin* or *repout* routine. Refer to the *flushtlb* manual page for more information.

Parameters

dev_addr is the physical I/O address where reading begins.

kv_addr is the kernel virtual address where the data will be stored. It must be the base address of an array large enough to hold *cnt* items.

The *cnt* parameter is one of the following values:

- The number of bytes to be read by *repinsb*
- The number of 16-bit words to be read by *repinsw*
- The number of 32-bit words to be read by *repinsd*

For the *repout* routines, *dev_addr* is the physical I/O address where writing begins.

kv_addr is the kernel virtual address where the data is stored. It must be the base address of an array of size *cnt* items.

The *cnt* parameter is one of the following values:

- the number of bytes to be written for *repoutsb*
- the number of 16-bit words to be written for *repoutsw*
- the number of 32-bit words to be written for *repoutsd*

Examples

The following three examples demonstrate how to use the *repins* routines. In each case the variable *dev_addr* would need to be assigned an appropriate value (I/O address) before being used as a parameter. In these examples *kv_addr* specifies an array of memory declared locally by the device driver but it could be any kernel virtual address.

```
/* repinsb */
char kv_addr[50];
repinsb(dev_addr, (caddr_t) kv_addr, 50);

/* repinsw */
short int kv_addr[50];
repinsw(dev_addr, (caddr_t) kv_addr, 50);

/* repinsd */
int kv_addr[50];
repinsd(dev_addr, (caddr_t) kv_addr, 50);
```

The following three examples demonstrate how to use the *repout* routines. In each case the variable *dev_addr* would need to be assigned an appropriate value (I/O address) before being used as a parameter. In these examples *kv_addr* specifies an array of memory declared locally by the device driver, but note that *kv_addr* can be any kernel virtual address.

```
/* repoutsb */
char kv_addr[50];
repoutsb(dev_addr, (caddr_t) kv_addr, 50);

/* repoutsw */
short int kv_addr[50];
repoutsw(dev_addr, (caddr_t) kv_addr, 50);

/* repoutsd */
int kv_addr[50];
repoutsd(dev_addr, (caddr_t) kv_addr, 50);
```

See Also

flushtlb(K), inb(K), inw(K), ind(K), copyin(K)

select: selsuccess, selfailure, selwakeup

kernel routines supporting select(S)

Syntax

#include "sys/select.h"

void selsuccess( )

void selfailure( )

void selwakeup(proc-ptr, flags)
struct proc *proc-ptr;
char flags;

Description

The *select*(S) system call code sends a unique I/O control command to the driver to ask whether or not a condition is satisfied for reading, writing, or exceptional circumstances. The *mode* argument to the *xxioctl* call indicates the condition being selected, and has these values which are defined in **sys/select.h**: SELREAD, SELWRITE, and SELEXCEPT. All drivers that support *select* must implement the IOC_SELECT ioctl with code similar to that displayed in the example section in this manual page.

Drivers supporting *select*(S) must include global declarations like the following:

```
#include "sys/select.h"
extern int selwait;
struct xx_selstr
{
        struct proc *read;
        struct proc *write;
        struct proc *except;
        char flags;
}
xxselstr[NUM_MINOR_DEVS];
```

A driver uses *selfailure* to indicate that the condition which the user has selected is not true, and the process should block.

A driver uses *selsuccess* to indicate that the condition which the user has selected is true, and the process should not block.

A driver uses *selwakeup* to indicate that the condition the user selected which was not initially satisfied, is now true. The process should now be awakened.

Parameters

procp is a pointer to a process table entry which is found in the *xxselstr* data structure. Each time a process selects a condition that is not immediately satisfied, a pointer to the process is stored in the data structure. This pointer is passed to the *select*(S) system call by the *selwakeup* call.

flags is a byte used to indicating if multiple processes are colliding by selecting the same condition. Whenever more than one process selects a condition, the driver must set the correct collision bit. The three bits defined in **sys/select.h** are: READ_COLLISION, WRITE_COLLISION, and EXCEPT_COLLISION, for selecting for reading, writing, and exceptional conditions, respectively.

Examples

In the first example, a process issues the *select*(S) system call (read case only): Note that the examples can be replicated identically substituting write and except for all occurrences of read.

```
/*
* This routine from driver xx
* handles a select for read request.
* Note all requests come from
* the select system call individually.
*/

xxioctl(dev, cmd, mode, arg)
dev_t dev;
int cmd, mode, arg;
{
        ...
        if ( cmd == IOC_SELECT )
                {
                switch (mode)
                {
                case SELREAD:   xx_selread(dev);
                break;

                /*
                * likewise for SELWRITE and SELEXCEPT
                */
                }
                return;
                }
        ...
/*
* Normal ioctl processing
*/

}

xx_selread(dev)
dev_t dev;
{
        extern void selsuccess();
        extern void selfailure();
        struct proc *procp;
        struct xx_selstr *ptr = &xxselstr[dev];

        if ( xx_condition_is_satisfied_for_read[dev] )
        {
                selsuccess();
                return;
        }

        /*
         * Condition is unsatisfied; process will block.
         */

        procp = ptr->read;
        if ( procp && procp->p_wchan == (char*) &selwait )
                ptr->flags |= READ_COLLISION;
        else
                ptr->read = u.u_procp;
        selfailure();
}
```

In the next example, the process has selected a condition and is blocked. Then, the condition becomes satisfied (read case handled):

```
xxintr(level)
{
        ...

        /*
        * Driver first notices that the condition is now
        * satisfied and computes minor dev
        */

        xxwakeread(dev);
        ...
}
```

```
xxwakeread(dev)
dev_t dev;
{
        struct xx_selstr *ptr = &xxselstr[dev];

        /*
        * If a proc has selected the condition, awaken it.
        */

        if ( ptr->read )
        {
                selwakeup(ptr->read, ptr->flags
                                & READ_COLLISION);
                ptr->read = (struct proc *) NULL;
                ptr->flags &= ~READ_COLLISION;
        }
}
```

scsi: scsi_get_gen_cmd, scsi_getdev, scsi_mkadr3, scsi_s2tos, scsi_s3tol, scsi_stok, scsi_stol, scsi_swap4

SCSI routines

Syntax

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

int
scsi_get_gen_cmd(request_ptr, command, arrayp, nelements)
REQ_IO request_ptr;
int command, nelements;
paddr_t *arrayp;

DEVCFG
scsi_getdev(unit, config-tbl, dev_open, name)
int unit;
DEVCFG *config-tbl;
int (*dev_open)( );
char *name;

int
scsi_mkadr3(str, adr)
char str[ ];
char adr[ ];

short
scsi_s2tos(adr)
char *adr;

long
scsi_s3tol(adr)
char  adr[ ];

caddr_t
scsi_stok(adr)
char  adr[ ];
```

```
long
scsi_stol(adr)
char adr[ ];

int
scsi_swap4(adr)
char *adr;
```

Description

These routines are used when converting between different SCSI addressing schemes, and for miscellaneous tasks. *scsi_get_gen_cmd* is used to fill a command block; *scsi_getdev* gets a SCSI device number. Refer to Chapter 7, "Writing a SCSI Driver" for conceptual information on the use of the *scsi*(K) routines. The conversion routines are:

Routine	**Source**				**Destination**			
scsi_mkadr3(str, adr)	*adr*				*str*			
Makes 3-byte address		2	1	0		2	1	0
		a	b	c		c	b	a
short = scsi_s2tos(adr)	*adr*				*short*			
Converts 2 bytes to a short			1	0			1	0
			a	b			b	a
long = scsi_s3tol(adr)	*adr*				*long*			
Converts 3 bytes to a long		2	1	0	3	2	1	0
		a	b	c	0	c	b	a
kernel = scsi_stok(adr)	*adr*				*kernel*			
Converts 3 bytes to a		2	1	0	3	2	1	0
kernel address		a	b	c	0	c	b	a
long = scsi_stol(adr)	*adr*				*long*			
Converts 4 bytes to a long	3	2	1	0	3	2	1	0
	a	b	c	d	d	c	b	a
scsi_swap4(adr)	*adr*				*adr*			
Swaps 4 bytes	3	2	1	0	3	2	1	0
	a	b	c	d	d	c	b	a

Parameters

scsi_get_gen_cmd	*request_ptr* is a pointer to an instance of the *scsi_io_req* structure (SCSI I/O request block). *command* is: TEST_CMD, SENSE_CMD, or INQUIRY_CMD which are defined in *scsi.h*. *arrayp* is a pointer to an array containing the arguments to *command*. *nelements* indicates the number of elements in the array pointed to by *arrayp*.
scsi_getdev	*unit* is a logical unit number, such as a device number. *config-tbl* is a pointer to an configuration table (an array) in which device information is stored that is used to build a block or character device switch table entry. The configuration table is an instance of the *scsi_dev_cfg* structure defined in **scsi.h** and must be NULL terminated. The *index* field of this structure is filled in by the operating system and contains the major number of the device. The *dev_open* argument to *scsi_getdev* is the address of the *xxopen* routine for your driver. *name* is the address of an array containing the name of the disk. For example, for the SCSI disk driver supplied with your system, "Sdsk" is used.
scsi_mkadr3	*str* is a pointer to a three-byte array; *adr* points to a three-byte array for the address to be created. The most significant byte of *adr* is not used.
scsi_s2tos	*adr* in a pointer to a two-byte array.
scsi_s3tol	*adr* in a pointer to a three-byte array.
scsi_stok	*adr* in a pointer to a three-byte array.
scsi_stol	*adr* in a pointer to a four-byte array.
scsi_swap4	*adr* in a pointer to a four-byte array.

Return Value

scsi_get_gen_cmd has no return value under normal conditions, but -1 can be returned if *command* is not one of the three permissible values.

scsi_getdev returns a pointer to the configuration table entry for the device or NULL if the device cannot be found in the configuration table. In addition to returning NULL, the following message is displayed on the console and stored in the *putbuf* array in memory:

`WARNING: No configuration information for` *device name*

Where *device* is the major device number and *name* is the argument to the routine.

scsi_s2tos returns a short integer. *scsi_s3tol* returns a long integer. *scsi_stok* returns a kernel address. *scsi_stol* returns a long integer.

seterror

sets u.u_error with error code

Syntax

#include "sys/errno.h"

void
seterror(errno)
int errno;

Description

This routine sets **u.u_error** to the value specified in *errno*. Possible values for *errno* are defined in **sys/errno.h**. This routine can only be called from user context and not during system initialization.

Typically, the following error codes are used in drivers:

Value	Description
EACCES	Permission denied
EAGAIN	No more processes
EBUSY	Mount device busy
EFAULT	Bad address
EINTR	Interrupted system call
EINVAL	Invalid argument
EIO	I/O error
ENODEV	No such device
ENOENT	No such file or directory
ENOMEM	Not enough core
ENOSPC	No space left on device
ENXIO	No such device or address
EPERM	Not super-user

signal

sends a signal to a process

Syntax

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

int
signal(pgrp, signum)
int pgrp, signum;
```

Description

The *signal* routine sends the specified signal, *signum*, to all processes in the process group identified by *pgrp*.

Parameters

The *pgrp* argument is an integer that specifies the process group number. At task time *pgrp* is one of the two equivalent integers **u.u_procp->p_pgrp** or **u.u_ttyp->t_pgrp**. If you wished to be able to terminate a process group at interrupt time you would need to store the *pgrp* ID in a global variable.

The *signum* argument is an integer that specifies the signal to be sent.

See Also

psignal(K)

sleep

suspends processing temporarily

Syntax

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

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

Description

sleep suspends task time processing in a driver. Its behavior and functionality is not at all like that of the *sleep*(S) system call. For a temporary halt in the execution of your driver use the *delay*(K) routine. *sleep* should be used when it is necessary for the driver to wait until a resource is available or an I/O request has completed before continuing task time execution. It is not guaranteed that when *sleep* returns, the event or resource that the driver has been waiting for will have occurred.

Notes

This routine must not be called at interrupt time or from the *xxinit* routine.

Parameters

address is a number the system uses for identifying the sleeping process in the process table. This number should be chosen so that it is unique to those processes put to sleep by your driver. A good method for deriving a unique number is to use the address of a global variable that has been declared in your driver. Addresses of static variables are not guaranteed to be unique. During debugging it is useful for the device driver writer to display this number, since it is possible to use the shell command **ps -el** to identify which processes are sleeping in your driver by examining the values reported in the WCHAN column.

priority determines the priority of the process when it awakens. This value is used by the scheduler to determine execution order in the run queue. A lower priority places the process nearer the top of the run queue. In addition the value of *priority* is also used to determine whether the sleeping process can be interrupted by a software signal. If *priority* is less than PZERO, then the sleeping process cannot be interrupted by a software signal. A process should only sleep at a priority less than PZERO if it is guaranteed that the event it is waiting for will occur within a short time. In general, processes should sleep at priorities greater than PZERO so that users are able to force termination of their processes by a software signal if an error occurs.

Return Values

sleep returns 0 (zero) if a *wakeup*(K) routine has been called using the same *address* that was specified in the *sleep* call, or it returns 1 if the *priority* used has been ORed with PCATCH and the sleeping process has been sent a software signal.

Note

If *priority* has not been ORed with PCATCH and the sleeping process is interrupted by a software signal, *sleep* will not return control to the device driver. Instead sleep will *longjmp*(K) back to the process state just after the system call was made. The system call invoked by the user process will return -1 and *errno* will be set to EINTR. If the device driver has set flags or temporarily allocated memory that should be cleared or freed when *sleep* is interrupted, OR the PCATCH constant into *priority*. This causes control to return to the driver on a signal. The driver should then restore any temporary resources it was using, set **u.u_error** to EINTR, and return -1. See the example following the discussion of the *wakeup* routine for more information.

See Also

timeout(K), wakeup(K)

spl: spl0, spl1, spl2, spl3, spl4, spl5, spl6, spl7, splbuf, splcli, splhi, splni, splpp, spltty, splx

block/permit interrupts

Syntax

```
int         int         int          int
spl0()      spl4()      splbuf()     splpp()

int         int         int          int
spl1()      spl5()      splcli()     spltty()

int         int         int
spl2()      spl6()      splhi()

int         int         int
spl3()      spl7()      splni()


int
splx(previous)
int previous;
```

Description

In many drivers, a need exists to protect sections of code from an interrupt occuring and causing a context switch. Protecting code sections ensures the integrity of the kernel and its data structures. The *spl1*, *spl2*, *spl3*, *spl4*, *spl5*, *spl6*, *spl7*, *splbuf*, *splcli*, *splni*, *splpp* and *spltty* routines prevent specific levels of interrupts from occuring.

The following table describes each of the *spl* routines:

Routine	IPL	Description
spl0	0	Permit all interrupts to occur
spl1	1	Prevent interrupts from context and process switches
spl2	2	Prevent priority level 2 interrupts
spl3	3	Prevent priority level 3 interrupts
spl4	4	Prevent character device interrupts
spl5 *splcli* *splpp*	5	Prevent interrupts from character devices from a ports card, and from tty devices.
spl6 *splbuf* *splni*	6	Prevent interrupts from block devices, network devices, and the clock.
spltty	7	prevent tty interrupts (and the clock)
spl7 *splhi*	8	prevent all interrupts
splx	-	restore interrupt level to a former level

IPL stands for interrupt priority level. The IPL value for a device is set on its card.

spl0 permits all interrupts to occur. *splx* restores the interrupt level to that specified by its argument *oldspl*. Use of *spl0* is not encouraged because by restoring all interrupts, you may undo a previously set level by a kernel process or routine calling your driver. Use *splx* whenever possible to restore a previously set level. When coding an interrupt routine, only use *splx*.

The *spl5* and *splcli* routines prevent interrupts associated with tty devices and those that use character lists (*clist*s) to buffer data. *spl5* is provided for backward compatibility only. If you are writing a serial device driver, use *splcli* whenever possible. *splcli* should be used to protect critical sections of code which manipulate *clist* structures or pointers. It is possible that a device driver's *xxpoll* routine will preempt another driver while it is manipulating *clists*. If your *xxpoll* routine manipulates *clist* structures, you should exercise care to make sure that your routine was not entered at an *spl* level higher than 5. Otherwise you may corrupt the kernel *cfreelist*. Do not manipulate *clists* in an interrupt routine. It is not necessary to use *splcli* before calling any of the *cblock* or *clist* routines (the routines on the *getc*(K) and *putc*(K) manual pages) because these routines raise the system priority level before entering their critical sections and then restore it to its previous value before they return. It is only necessary for you to use *splcli* if you are directly manipulating fields in a *clist* structure or the freelist. You should only do this if you have extensive experience with character device drivers.

The *spl6*, *splni*, and *splbuf* routines mask all interrupts except for those from the serial device. These routines prevents block device interrupts from occuring. These routines block the system clock and should be used sparingly because if the code it is protecting takes longer than two clock ticks to execute, the system clock is degraded.

The *spltty* routine blocks interrupts from a serial device.

The *spl7* and *splhi* routines disable all interrupts. Use this routine only for extremely short periods when updating critical data structures that could be accessed by a high priority device. These routines also block the system clock and should not be allowed to execute longer than a single clock tick.

The *splx* routine restores a previously set interrupt priority level.

Parameters

The integer *oldspl* specifies a previous *spl* level, it should only be set by the return value of a previous *spl* routine.

Warnings

The interrupt priority level in an interrupt routine must not be dropped below the level at which the interrupt occurred or the stack may become corrupted causing a system panic or loss of data. Use of *spl0* is not recommended as it can lower previously set priority levels used by other kernel processes. Always store the previous priority level returned by the *spl* call and use *splx* to restore the previousl level at the end of your critical code section.

Return Value

The previous *spl* value is returned. It should be saved and used to restore the *spl* level by a subsequent call to *splx*.

The *spl7* and *splhi* routines block all interrupts. While these routines are in effect, the following happens:

- Clock interrupts do not occur.
- Characters are not echoed back to the console.
- The *capslock*, *numlock*, and *scrolllock* indicators do not work.
- Multi-screens cannot be switched.
- If your driver hangs during this interval, the machine must be power-cycled to regain control.

Notes

Use of *spl6*, *spl7*, and *splhi* can cause the software clock to lose time and prevents other device drivers' *xxpoll* routines from being called. This may have an unpredictable effect on the behavior of other device drivers that require periodic execution of their *xxpoll* routines.

Example

The following code fragments show how to check an *spl* level in an *xxpoll* routine and demonstrate the use of *splcli* in a task time routine.

```
/*
 * This macro tells us if the previous spl level was "lev"
 * or higher. PS_PRIMASK is defined in sys/param.h.
 */
#define ATSPL(lev,ps) ((ps) >= lev)

xxpoll(ps)
{
/*
 * If we were at spl5 or higher before the clock tick, leave!
 */
        if (ATSPL(5,ps)) {
                return;
        }
/*
** end xxpoll fragment.
*/

/*
 * xxread(), xxwrite(), xxopen(), xxclose, and
 * xxioctl() are all examples of task time routines
 */
xxread(dev)
int dev;
{
    int oldspl;

    /* set new spl and save old level in oldspl */
    oldspl = splcli();
    .
    .    /* perform clist operations */
    .
    /* restore saved spl */
    splx(oldspl);
    .
    .
    .
/*
** end task time fragment.
*/
}
```

sptalloc

allocates temporary memory or maps a device into memory

Syntax

#include "sys/immu.h"

caddr_t
sptalloc(pages, mode, base, flag)
int pages, mode, base, flag;

Description

The *sptalloc* routine is used to obtain temporary memory for use by device drivers, or to map a device into memory for memory mapped I/O. Memory is obtained from the system's virtual memory pool. When the driver is through with the memory, the memory should be released via *sptfree*(K). The *sptalloc* routine returns a virtual address usable by any kernel or driver routine.

Memory allocated is virtually contiguous but not physically contiguous. The memory allocated is never swapped out, and it only belongs to the driver that allocated it until the memory is freed with *sptfree*(K).

The usual way to call *sptalloc* is as follows:

```
        sptalloc(pages, PG_P, 0, 1);
```

Where *pages* are the number of requested pages, *mode* indicates "page present," *base* (zero) indicates that requested memory is taken from the kernel memory pool, and *flag* (1) indicates to return immediately if memory is not available.

To use *sptalloc* for accessing memory mapped I/O, use *sptalloc* as shown for an imaginary device being installed at physical address 0xB8000:

```
        sptalloc(1, PG_P, 0xB8000, 1);
```

Although *vasbind*(K) provides a more generalized method of sharing memory between the kernel and a user process, *sptalloc* with *mode* set to PG_P | PG_RW | PG_US may be used instead of *vasbind*, but the results are different.

A mapping performed with *vasbind* creates a region of memory shared only between the kernel and a specific user process. *sptalloc* creates a mapping accessible by the kernel and *all* processes. However, only those processes that have been told the virtual address returned from *sptalloc*, will know the address at which to access the memory.

Parameters

pages the number of requested pages

mode page descriptor table entry field mask. Possible values are defined in **sys/immu.h** and are:

- PG_P — page-present bit. This flag must be present for driver use. PG_P causes the *present* bit to be set in the page table entry. The CPU uses the present bit to differentiate between pages that have to be faulted in and pages that are already there.

- PG_RW — make segment usable for either reading or writing. If this flag is not ORed into *mode*, than the segment is read-only. This flag only has meaning when used with PG_US to indicate if a user can access the segment for both reading and writing. (Kernel processes can read or write any present page whether write access is "permitted' or not.)

- PG_US — identify owner of memory. If ORed in, memory is allocated for a user process. if omitted, memory is for a kernel process. If selected, any user process can access the page. Use with PG_RW if write permission is required; without PG_RW, the page is read-only. To use this capability, a driver must pass the return value from *sptalloc* back to the user process for it to "know" where the memory is, but this doesn't limit its use to that process.

base set to 0 (zero) to allocate kernel memory, or set to an physical address pointing to previously allocated memory elsewhere.

flag	Set to 1 to return immediately if memory is not available. Set to 0 (zero) to sleep until memory is available. If only one page is being requested, and memory is not available, sleep occurs however *flag* is set. When sleeping is requested, *sptalloc* sleeps with a priority of 0 (zero) and is not affected by signals.

Warning

Because *sptalloc* may sleep, do not use at interrupt time (from the *xxintr* routine).

Return

This routine returns the kernel virtual address of the memory allocated. NULL is returned if map space is not available. The size of the map is determined by the constant *sptmap* which is configurable using the Link Kit.

See Also

sptfree(K), vas(K)

sptfree

releases memory previously allocated with sptalloc

Syntax

void
sptfree(va, npages, freeflg)
char *va;
int npages;
int freeflg;

Description

The *sptfree* routine frees memory obtained from *sptalloc*(K). The arguments are the pointer returned by *sptalloc*, the size of the memory (same as passed to *sptalloc*) and a flag which denotes whether you want this freed memory to go back into the free page list. For drivers which use this to free memory obtained from *sptalloc*, the flag must always be 1.

For example, to release the memory obtained by a call to the *sptalloc* routine, and free it completely, use the following statement:

```
sptfree( va, npages, 1 );
```

Parameters

The argument *va* is the virtual address returned from a previous call to *sptalloc*.

The value of *npages* is the number of pages to free. This should be the same number of pages allocated by a previous call to *sptalloc*.

The argument *freeflg* indicates whether to actually free the memory pages or not. If *freeflg* is not set, the memory pages are not freed. This is used, for example, when freeing memory-mapped I/O space.

See Also

sptalloc(K)

subyte

stores a character in user data space

Syntax

int
subyte(addr, val)
unsigned char *addr, val;

Description

The *subyte* routine stores one character in the user's data space. If you are storing data in **u.u_base**, consider using *passc*(K) in that **u.u_count, u.u_offset,** and **u.u_base** are updated for you, and error handling is provided via **u.u_error**. If any chance exists that a -1 may be contained in the data that you are storing in user space, use *copyout*(K) instead of *subyte* (or *passc*). A -1 in the data is usually associated as an error condition.

This routine must not be called from an interrupt or *xxinit* routine.

Parameters

The argument *addr* is a pointer to the byte to be stored in the user's data space.

The argument *val* is the value to be stored.

See Also

fubyte(K), fuword(K), suword(K), passc(K)

suser

determines if current user is the super-user

Syntax

int
suser( )

Description

The *suser* routine determines whether the user associated with the currently executing process is the super-user. This can be useful, for example, in determining whether special device operations (such as disk formatting) are allowed.

Return

suser returns 0 if the current user is not the super-user and 1 if the user is the super-user.

suword

stores a 32-bit word in user data space

Syntax

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

Description

The *suword* routine stores one 32-bit word in the user's data space.

Parameters

The argument *addr* is a pointer to the beginning address in user space. (The address does not have to be word-aligned.)

The argument *val* is the value to be set.

This routine must not be called from an interrupt or *xxinit* routine.

See Also

fubyte(K), fuword(K), subyte(K)

timeout, untimeout

schedules a time to execute a routine

Syntax

```
int
timeout(routine, arg, clock_ticks)
int (*routine) ( );
caddr_t arg;
int clock_ticks;

void
untimeout(id)
int id;
```

Description

timeout schedules a routine to be executed at a specific time in the future. *timeout* returns an integer identification number. *untimeout* cancels a *timeout* request using the identfication number returned from *timeout*.

Note

This routine can be used in an interrupt routine only if the interrupt priority level does not block the clock interrupt. Do not use if the level is at *spl6, spl7, splhi, splni* or *spltty*.

Parameters

timeout has the following arguments:

routine	a routine to be executed after the specified number of *clock_ticks* has elapsed.
arg	passed as a parameter to *routine*.
clock_ticks	the number of clock ticks to wait before calling *routine*.

The argument to *untimeout* is the integer identification number returned from the *timeout* call that you wish to cancel.

Notes

timeout should normally only be used at task time, however it can be used in an *xxinit* routine with this caveat. Since the clock interrupts may not be enabled before your *xxinit* routine is called, the timeout may not elapse when specified, but will elapse no later than (*time_when_clock_started* + *clock_ticks*) times the length of one clock tick.

Example

timeout(K), *sleep*(K) and *wakeup*(K) can be combined to provide a ``busy, wait'' function. The following code sample illustrates this possible functionality:

```
#define PERIOD 5          /* 5 clock ticks */
#define BUSYPRI   (PZERO -1)  /* arbitrary */

/* Declare routine to use in timeout(). */
int stopwait();

/* flag which is used to indicate */
/* whether to continue waiting.   */
int status;

int busywait() /* wait until status is non-zero */
{
        while (status == 0) {
                timeout(stopwait, (caddr_t) &status, PERIOD);
                sleep(&status, BUSYPRI);
        }
}

int stopwait(arg)
caddr_t arg;
{
        if ( /* what I am waiting for has happened */ )
                status = 1;
        else
                wakeup(arg);
}
```

A device driver should never loop while waiting for a status change unless the delay is less than 100 microseconds. Also, setting a *timeout* for fewer than three clock ticks may result in the *sleep* call happening after the *timeout* has occurred. This results in a permanent sleep condition (hang).

See Also

sleep(K), wakeup(K), delay(K)

ttiocom

interpret tty driver I/O control commands

Syntax

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

int ttiocom(tp, cmd, arg, mode)
struct tty *tp;
int cmd, arg, mode;
```

Description

ttiocom sends an I/O control command to the tty device. Valid commands (the *cmd* argument to *ttiocom*) are:

- IOC_SELECT — determine if a character can be read from or written to a tty device without blocking (going to sleep in the process). *mode* can be SELREAD or SELWRITE. NOTE: IOC_SELECT must not be called from an interrupt routine and *sleep* must not be called just prior to calling this I/O control command. IOC_SELECT calls *ttselect*.

- IOCTYPE — return the name of the last I/O control command called. **u.u_rval1** is set to the value of TIOC. IOCTYPE must not be called from an interrupt routine.

- TCSETAF, TCSETAW, TCSETA, TCGETA, TCSBRK, TCXONC, TCFLSH — explained on the *termio*(M) manual page. TCSETAW and TCSETAF call *ttywait*. TCSETAF calls *ttyflush*. TCSETA calls *ttioctl* when opening a new line discipline and when changing the value of the line discipline flag, **t_lflag**. TCSBRK calls *ttywait*. TCXONC calls the driver *xxproc* routine with varying arguments depending on the *arg* argument to *ttiocom*. TCGETA sets **u.u_error** to EFAULT if a paging error occurs while trying to return the requested **tty** structure. TCXONC sets **u.u_error** to EINVAL if *arg* is not 0, 1, 2, or 3. TCFLSH sets **u.u_error** to EINVAL if *arg* is not 0, 1, or 2. TCSETA sets **u.u_error** to EFAULT if the **tty** structure cannot be set, or to EINVAL if the requested line discipline is less than zero or greater than the maximum. TCFLSH calls *ttyflush*. *xxproc* is called by TCXONC as follows:

arg value	*xxproc argument*
0 (zero)	T_SUSPEND
1	T_RESUME
2	T_BLOCK
3	T_UNBLOCK

- FIORDCHK — check to see if characters are waiting to be read. 1 is returned if characters are waiting in **t_canq**. If ICANON is set, it is also possible for 1 to be returned when characters are not in **t_canq**, but there are characters in **t_delct**. If there are no characters in **t_canq** and ICANON is not set, and if there are characters in **t_rawq**, 1 is returned. If none of the queues have characters, 0 (zero) is returned. FIORDCHK causes *ttrdchk* to be called.

- XCSETAW — wait for the universal asynchronous receiver/transmitter (UART) to empty (waits 11 bit times depending on the terminal's baud rate). XCSETAW is a POSIX *termio* extension.

- XCSETAF — wait until the UART empties and then flush all read and write buffers (calls *ttyflush*). XCSETAF is a POSIX *termio* extension.

- XCSETA — set terminal parameters from the **tty** structure specified by the *arg* argument to *ttiocom*. XCSETA is a POSIX *termio* extension.

- XCGETA — get terminal parameters from a terminal's **tty** structure and put into the **tty** structure specified by the *arg* argument to *ttiocom*.

Parameters

tp	Pointer to an instance of the **tty** structure for a tty device
cmd	I/O control command passed through from the user program
arg	Argument to the I/O control command, also passed through from the user program
mode	Indicates the mode by which the file was opened. The modes are assigned by the kernel and are interpreted into flag values that are defined in **sys/file.h**. Possible values are FNDELAY, FREAD, FSTOPIO, FWRITE.

See Also

termio(M), canon(K), tty(K) (All other *tt* routines are described on the tty(K) manual page.)

vas: vasbind, vasmalloc, vasmapped, vasunbind

virtual address space memory routines

Syntax

```
int
vasbind(paddr, vaddr, nbytes)
paddr_t paddr;
caddr_t vaddr;
unsigned int nbytes;

caddr_t
vasmalloc(paddr, nbytes)
paddr_t paddr;
unsigned int nbytes;

caddr_t
vasmapped(paddr, nbytes)
paddr_t paddr;
unsigned int nbytes;

int
vasunbind(vaddr, nbytes)
caddr_t vaddr;
unsigned int nbytes;
```

Description

These routines allow a driver to map physical memory so that it can be read from or written to by both a driver and a calling user process. These routines are generally used to allow user processes to directly access video adapter memory. Memory that has been mapped using these routines is visible to the kernel and to a calling process. However, the mapping is not globally visible to all processes.

vasmalloc allocates virtual memory. Use this routine to obtain virtual address space that is not currently in use. *vasmalloc* can only allocate four megabytes of virtual address space on each call. Requests less than this amount are rounded up to four megabytes; requests larger than this amount cause an error to occur. *vasmalloc* returns an address to virtual user memory; no actual physical memory is allocated by this routine. The *nbytes* argument can be specified as 1 to allocate four megabytes, but 0 (zero) or not specifying this argument is not permissible.

vasbind binds a specified virtual address to a physical address. This routine ensures that a problem will not occur with the bound memory being swapped out and causing a page fault and panic in the kernel. Before using *vasbind*, call *vasmapped* to determine if memory has already been mapped for the calling process.

The physical address supplied to *vasbind* may be the address of an I/O address space, for example, a memory-mapped I/O address. Or specify -1 to request that the memory be allocated from the kernel free memory pool.

When *vasbind* completes, the driver must pass the virtual address back to the user process using *copyout*(K) or another similar routine. Calls to *vasbind* must not specify an address in the text, data, or shared data segments of a user process.

The upper limit for user virtual memory is set in the KVBASE constant (defined in **sys/immu.h**); above KVBASE is the kernel virtual address space. The virtual address supplied to *vasbind* must be in user virtual memory (below KVBASE), and must not be in use by the current process.

vasmapped determines if a mapping is already in place.

vasunbind undoes a mapping.

Notes

These routines cannot be called from a driver's interrupt routine (*xxintr*).

Parameters

nbytes	number of bytes of memory to allocate, bind, or unbind. For *vasmalloc*, *nbytes* can be specified as 1 to allocate four megabytes, but 0 (zero) or not specifying this argument is not permissible.
paddr	Physical address at which the specified virtual address is to be bound. When calling *vasbind*, *paddr* can be set to -1 to indicate that the requested user virtual memory is to be allocated.
vaddr	Virtual address to bind or unbind to or from physical memory

Return Value

vasbind returns -1 if an error occurs or if an error is found in **u.u_error**. *vasmalloc* returns a virtual address. *vasunbind* returns -1 if an error is found in **u.u_error**, or if the virtual address couldn't be found. *vasmapped* returns the virtual address at which the supplied physical address is bound, or 0 (zero) is physical address is not bound.

See Also

sptalloc(K), copyout(K)

tty: ttclose, ttin, ttinit, ttiwake, ttopen, ttout, ttowake, ttread, ttrdchk, ttrstrt, ttselect, tttimeo, ttwrite, ttxput, ttyflush, ttywait

tty driver routines

Syntax

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

int
ttclose(tp)
struct tty *tp;

int
ttin(tp, code)
struct tty *tp;
int code;

int
ttinit(tp)
struct tty *tp;

int
ttioctl(tp, cmd, arg, mode)
struct tty *tp;
int cmd, arg, mode;

int
ttiwake(tp)
struct tty *tp;

int
ttopen(tp)
struct tty *tp;


int
ttout(tp)
struct tty *tp;
```

```
int
ttowake(tp)
struct tty *tp;

int
ttread(tp)
struct tty *tp;

int
ttrdchk(tp)
struct tty *tp;

int
ttrstrt(tp)
struct tty *tp;

int
ttselect(tp, rw)
struct tty *tp;
int rw;

int
tttimeo(tp)
struct tty *tp;

int
ttwrite(tp)
struct tty *tp;

int
ttxput(tp, ucp, ncode)
struct tty *tp;
int ncode;
union {
      unsigned short ch;
      struct cblock *ptr;
} ucp;

int
ttyflush(tp, rdwrt)
struct tty *tp;
int rdwrt;

int
ttywait(tp)
struct tty *tp;
```

Description

The routines are:

ttclose called by the line discipline zero *l_close* routine to remove access to a tty device from the process that called it. *ttclose* disables ISOPEN from the **t_state** member of the **tty** structure, calls *ttioctl* with the LDCLOSE argument, and disables XCLUDE from the **t_lflag** member of the **tty** structure.

ttin called by the line discipline zero *l_input* routine to get characters from a TTY device. *ttin* is called in a driver's interrupt routine to process and move characters from **t_rbuf** to the raw character queue, **t_rawq**. *ttin* processes the *termio*(M) **c_cc** values of VINTR, VQUIT, VSUSP, VSWTCH, VEOL, VERASE, VKILL, and VEOF. In addition, *ttin* checks that ICANON is set when processing the **c_cc** VMIN and VTIME values. *ttin* performs input escape mapping for internationalization character processing. If the *code* argument to *ttin* is L_BREAK, *ttin* sends the SIGINT signal to all associated processes. *ttin* then calls *ttyflush* to release both read and write buffers, and returns if no characters are found in **t_rbuf**. If either ECHO is set or after processing international character mapping, *ttin* calls the driver's *xxproc* routine with T_OUTPUT set. *xxproc* can also be called with T_SWTCH set if VSWTCH is enabled and if NOFLSH is disabled. *ttin* calls these other tty routines: *ttyflush*, *ttxput*, *tttimeo*, and *ttiwake*.

ttinit initializes the **tty** structure for line discipline 0 (zero). The following members of the **tty** structure are initialized:

- **t_line** — line discipline index is set to 0 (zero)
- **t_iflag** — terminal input control is set to 0 (zero)
- **t_oflag** — terminal output control is set to 0 (zero)
- **t_lflag** — line discipline terminal control is set to 0 (zero)

- **t_cflag** — terminal hardware control is set to these values: 1200 baud (SSPEED variable), 8 bits (CS8 variable), enable receiver (CREAD variable), hang up on last close (HUPCL variable)

ttioctl used to allocate, deallocate, or move the contents of terminal buffers. Called through the *l_ioctl* function of the line switch table. The *cmd* argument to *ttioctl* has the following values:

- LDOPEN — allocate a receive buffer, initialize several **t_rbuf** members (**c_ptr, c_count,** and **c_size**), and call the driver's *xxproc* routine with T_INPUT as the argument.

- LDCLOSE — call the driver's *xxproc* routine with the T_RESUME argument; call *ttywait* to wait for data to clear the UART; and call *ttyflush* to empty the **t_canq** and **t_rawq** buffers, as well as the contents of **t_rbuf**.

- LDCHG — move contents of the raw queue, **t_rawq**, to the canonical queue, **t_canq**. LDCHG only works if ICANON is enabled.

ttiwake called from an interrupt routine to wake up any processes that are asleep waiting for characters to appear in the raw input queue, **t_rawq**. *ttiwake* only works when IASLP is set in the **t_state** field of the **tty** structure. *ttiwake* disables IASLP.

ttowake called from an interrupt routine to wake up any processes that are asleep waiting for characters to appear in the output queue, **t_outq**. *ttowake* only works when OASLP is set in the **t_state** field of the **tty** structure. *ttowake* disables OASLP.

ttread called from a driver's *xxread* routine indirectly through the *l_read* line discipline switch function. *ttread* conveys characters processed by *canon* from the canonical queue, **t_canq**, to the user process.

ttrdchk returns a non-zero value if there are characters to be read. If carrier is present (CARR_ON is enabled in **t_state**), *ttrdchk* tests **t_canq** for characters. **t_canq** is the queue for characters that have been processed by the *canon* routine. If characters are present, a 1 is returned. If characters are not present, and *canon* is being used to process characters from the terminal (ICANON is set in **t_lflag**), then *ttrdchk* tests the delimiter count to see if a delimiter has been entered on the tty device. The delimiter count is held in **t_delct** in the **tty** structure. If a delimiter was received, 1 is returned. If characters are not present, but *canon* is not being used to process characters, then the raw character queue, **t_rawq**, is checked for characters. If characters are waiting in the raw queue, 1 is returned.

ttrstrt restart tty device output after a delay timeout. *ttrstrt* performs only one task; it calls the driver's *xxproc* routine with arguments of *tp* and T_TIME. (*tp* is the pointer to the *tty* structure passed through from the argument to *ttrstrt*.)

ttselect ensure that a read or write can be performed with no blocking. (If blocking occurs, the process will be put to sleep until the I/O can be satisfied.) Do not call *sleep*(K) before calling *ttselect*, and do not call *ttselect* from an interrupt routine. *ttselect* has two modes determined by its *rw* argument. These modes are SELREAD and SELWRITE.

tttimeo satisfy VTIME timing requirement for data input. *tttimeo* does not execute if ICANON is set, if VTIME or VMIN are not set, or if there aren't any characters to process in the raw queue, **t_rawq**. *tttimeo* calls *timeout*(K) for the time specified in VTIME times the number of ticks per second (Hz) divided by 10. While *timeout* is active, **t_state** is ORed with RTO and TACT. When the timing finishes, *ttiwake* is called to awaken any processes that are sleeping on the raw input queue.

ttwrite called from the line discipline *l_write* function call in a driver's *xxwrite* routine to copy data from the user program into the driver so that the data can be displayed on a tty device. If there is no carrier, *ttwrite* returns immediately. *ttwrite* requires user context and therefore cannot be called from an interrupt routine. If a paging fault occurs while attempting to get the data from user space, **u.u_error** is set to EFAULT. In the course of the copy operation, **u.u_base** and **u.u_count** are updated to reflect the amount of data transferred. *ttwrite* calls *ttxput* to write the data to the terminal.

Before data is fetched from user space, the number of characters in the output queue, **t_outq**, are checked to see if the count exceeds the high water mark. The high water mark is the point at which tty input is suspended so that the characters coming in don't overload the tty driver's ability to process them and echo them to output. If the count exceeds the high water mark, the driver's *xxproc* routine is called with the T_OUTPUT argument, and the process is put to sleep until the tty driver can handle I/O again. The process sleeps on the address of **t_outq** at a priority of TTOPRI (above PZERO) and can be awakened prematurely by a signal. If a signal is sent to the sleeping process, a *longjmp*(K) occurs, returning control to the calling user process and putting EINTR in **u.u_error** (*errno* in user space). If a write occurs in the background, SIGTTOU is sent to all processes in the current process group.

ttxput puts characters on the tty output queue (**t_outq**), adds delays, expands tabs, and handles carriage return and new line characters. *ttxput* is called from both base level to output characters to a tty device and from interrupt level for echoing characters read in from the tty device. If the queue escape character, denoted as QESC, is detected, it is put directly on the output queue. The next character after QESC is treated as a timer character if the octal value is greater than 200. The timer character is used in a *timeout*(K) call in *ttout*. If the character after QESC is less than 200, it is treated as an ordinary character to be output.

If **t_state** contains EXTPROC, but **t_lflag** does not contains XCASE, then no post processing is required. Additionally in this state, characters can be internationally mapped if requested, and shipped to the output queue for further processing. XCASE indicates that special characters should be ``escaped'' by being preceded with a backslash. Refer to *termio*(M) for a list of characters that are translated when XCASE is specified. International character mapping is requested if **t_mstate** is true. If XCASE processing is required, then international character mapping can also be requested, and processing for QESC is available (QESC handling is not available when no post processing is required).

ttxput processes characters with octal values greater than 200 as special characters and performs delay processing if **t_state** does not contain EXTPROC. If EXTPROC is set, than delay processing is assumed to be handled by an external process. Characters that may require translation and also need to indicate a delay are shown in the following table:

Octal Value	Description
0201	non-printing character
0202	backspace
0203	line feed
0204	tab
0205	vertical tab
0206	carriage return
0207	form feed

If delay handling is being provided by *ttxput*, then the delay is calculated based on the value of **t_oflag**. The *ttout* routine does the actual timed delay. *ttxput* outputs the QESC character as well as the timing value ORed with octal 0200.

ttyflush called to release character blocks to the free list from the write buffer, **t_outq**, or from the two read buffers, **t_canq** and **t_rawq**. The *rdwrt* argument to *ttyflush* should be ANDed with either FREAD to release read buffers, or with FWRITE to release the write buffer. When the blocks in the write buffer are released, the driver's *xxproc* routine is called with the T_WFLUSH argument. If **t_state** contains OASLP, any processes sleeping on the address of **t_outq** are awakened and OASLP is disabled, and if **t_state** contains TTIOW, then TTIOW is disabled and all processes sleeping on the address of **t_oflag** are awakened.

If the blocks in the read buffers are being released, then the driver's *xxproc* routine is called with the T_RFLUSH argument. If **t_state** contains IASLP, IASLP is disabled and all processes sleeping on **t_rawq** are awakened.

ttywait called to drain the contents of the universal asynchronous receiver/transmitter (UART). *ttywait* waits 11 bit times for the UART to empty of all data. The baud rate is taken from **t_cflag&CBAUD** and possible values are (in bits per second): 0.5, 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, or 38400. If upon entry to this routine, characters are in **t_outq**, or **t_state** contains BUSY or TIMEOUT, then **t_state** is ORed with TTIOW and the process is put to sleep on the address of **t_oflag** at TTOPRI. TTOPRI has a value above PZERO and therefore can be prematurely awakened by a signal. Should a signal occur, control returns to the user program and the EINTR error code is placed in **u.u_error** and *errno* in the user program.

Parameters

tp	a pointer to the **struct tty** data structure associated with the device being accessed.
code	used by *ttin* and can be set to L_BREAK to cause all processes associated with the terminal to be sent a SIGINT signal, and to have all character blocks on **t_outq, t_canq,** and **t_rawq** queues be released to the free list.
cmd, arg, or *mode*	used by *ttioctl* to determine which I/O control command is being requested.
rw	used by *ttselect* to determine whether the driver is testing for read or write access without blocking. Possible values are SELREAD or SELWRITE.
ucp, ncode	used by *ttxput*. *ucp* describes a union of a character or a pointer to a block of characters. *ncode* is a flag used to indicate which value *ucp* contains; if *ncode* is zero (0), *ucp* is a character; any other value indicates that *ucp* is a pointer to a **cblock**.
rdwrt	used by *ttyflush* to indicate which buffers to release to the free list. AND with FREAD to indicate that the **cblocks** in **t_canq** and **t_rawq** should be released; AND with FWRITE to indicate that the blocks in the **t_outq** should be released.

Notes

All routines on this manual page can be used only with character device drivers.

See Also

termio(M), canon(K), ttiocom(K)

video: DISPLAYED, viddoio, vidinitscreen, vidmap, vidresscreen, vidsavscreen, vidumapinit, vidunmap

supports video adapter driver development

Syntax

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

int
DISPLAYED(msp)
struct mscrn *mps;

int
viddoio(msp, arg, portlist)
struct mscrn *mps;
struct port_io_arg *arg;
struct portrange *portlist;

int
vidinitscreen(msp)
struct mscrn *mps;

caddr_t
vidmap(address, nbytes)
paddr_t address;
int nbytes;

int
vidresscreen(msp)
struct mscrn *mps;

int
vidsavscreen(msp)
struct mscrn *mps;

int
vidumapinit(base, size)
int base, size;

int
vidunmap(address, nbytes)
caddr_t address;
int nbytes;
```

Description

The descriptions of the routines are as follows:

DISPLAYED	Returns TRUE if the screen is displayed. DISPLAYED is a macro defined in **sys/vid.h**. The *msp* argument is a pointer to an instance of the multiscreen **mscrn** structure.
viddoio	Supports input and output I/O control commands for the adapter driver. The adapter driver passes the user's **port_io_arg** pointer, *arg*, and a list of I/O ports. The **port_io_arg** and **portrange** structures are documented in **sys/vid.h**. The *portlist* is terminated with a **portrange** count field of zero. The *msp* argument is a pointer to an instance of the multiscreen **mscrn** structure.
vidinitscreen	Initializes a multiscreen to 80 rows, 25 cols, white on black, and so on. The *msp* argument is a pointer to an instance of the multiscreen **mscrn** structure.
vidmap	Maps *nbytes* of physical memory *address* to virtual memory. A kernel data pointer to the virtual memory is returned. Unlike **memget**(K), no physical memory is actually added to the kernel address space: **vidmap** simply translates a physical address pointer to virtual.
vidresscreen	Restores the screen. Calls the adapter driver associated with a multiscreen to restore the screen. The *msp* argument is a pointer to an instance of the multiscreen **mscrn** structure.
vidsavscreen	Saves the screen. Calls the adapter driver associated with a multiscreen to save the screen. The *msp* argument is a pointer to an instance of the multiscreen **mscrn** structure.
vidumapinit	Returns a user physical address mapped to memory starting at virtual *base* extending for *size* bytes.
vidunmap	Unmaps *nbytes* of a previously mapped *address*. This routine is the reverse of *vidmap*.

vtop

convert a virtual address to a physical address

Syntax

#include "sys/types.h"

paddr_t
vtop(vaddr, proc-ptr)
char *vaddr;
struct proc *proc-ptr;

Description

vtop returns a physical address associated with the specified virtual address. Specify *proc-ptr* as either **u.u_procp** or as **bp->b_proc**, where *bp* is a pointer to the *buf* structure. *proc-ptr* can be omitted if *vtop* is called from user context. If *vtop* is called with one argument, or with **u.u_procp** as the *proc-ptr*, then *vtop* can only be called from user context and not from an interrupt routine.

Parameters

vaddr	virtual address to be translated
proc-ptr	pointer to a *proc* structure (described in *sys/proc.h*)

Return Value

Normally, a physical address is returned. -1 is returned if the specified address is incorrect and *vtop* is called from user context. 0 (zero) is returned if the specified address is incorrect and *vtop* is called from an interrupt routine. If the specified page of kernel memory is not present, a panic results during the *vtop* call with the following message:

```
PANIC: svirtophys - not present.
```

"*svirtophys*" is a subroutine called by *vtop*.

wakeup

wakes up a sleeping process

Syntax

```
int
wakeup(address)
caddr_t address;
```

Description

wakeup causes all processes which are sleeping at a wait channel equal to *address* to be taken off the sleep queue and placed on the run queue. When a process is awakened, the call to *sleep*(K) returns a value of zero. It is still necessary to see whether the event being slept on has occurred, as there is no guarantee that the resource being waited for is actually free.

Parameters

address should be the same value used in a previous invocation of the *sleep* routine. Since this number is not guaranteed to be unique and multiple processes may have have been awakened by any single invocation of the *wakeup* routine it is not guaranteed that the event being waited for has in fact occurred.

Return Value

wakeup does not return a useful value.

Example

The following code fragments demonstrate one possible use of *sleep* and *wakeup*. In this instance the driver *xxread* routine allocates a temporary storage area, queues an I/O transfer, and then puts the process to sleep. The *xxintr* routine is called when the device is ready to do the transfer. After the transfer is complete the *xxintr* routine executes a *wakeup*.

```
/*
 * declare variable which is used for address
 */
char my_address;
/*
 * First the xxread().
 */
xxread(dev)
int dev;
{
#define MYPRI    PZERO+15  /* PZERO is defined in sys/param.h */

        /* allocate temporary storage */
        .
        .
        .
        /* set flag to indicate I/O transfer is in progress */
        .
        /* start I/O transfer */
        .
        .
        /*
         * flag only indicates transfer is done if wakeup() has
         * been called by my xxintr().
         */
        while (/* flag indicates transfer is not done */) {
        /*
         * OR MYPRI with PCATCH to clean things up instead
         * of returning directly to user space with an error.
         * See longjmp(K) for more information.
        /*
        if (sleep(&my_address, MYPRI | PCATCH) == 1) {
                /* stop I/O transfer       */
                /* clear I/O transfer flag */
                /* free temporary memory   */
                u.u_error = EINTR;
                return(-1);
                }
        } /* only get past here when transfer is done */
        /* copy data from temporary storage to user address */
        .
        /* free temporary memory */
        return( /* number of bytes transferred */ );
}
```

```
/*
 * now for the xxintr()
 */
xxintr(interrupt)
int interrupt;
{
        /* check that transfer is complete */
        .
        .
        /* set flag to indicate transfer is complete */
        .
        /* wakeup sleeping process */
        wakeup(&my_address);
}
/*
 * Note that the preceding example does not take into
 * account the possibility that multiple user processes
 * may have queued requests for a single device.
 */
```

See Also

sleep(K), timeout(K), delay(K)

Index

8259 interrupt controller 6-9

A

AC_* values 5-21
AC_RESSCRN
  example 5-5
AC_SAVSCRN
  example 5-5
AC_SAVSZQRY
  screen memory allocation 5-7
adapctl routine
  video driver 5-21
Adapter structure 5-8
adapter_entry 7-7
Address conversion 2-36
AI_COLOR 5-16
AI_PRESENT 5-16
Alignment 6-2
allocb 9-8
ANSI parser 5-3
ANSI SCSI standard 7-2
apd_area 5-9
Appended writes
  block driver 3-8
  character driver 4-3
Assembly language guideline 2-39
Automatic variables 2-17, 2-18

B

Backup programs
  use of raw I/O 3-3
Bad tracking 7-16
b_addr 3-9
b_blkno 3-9
b_blkno Change A-4
B_BUSY 3-5
b_dev 6-12
bdevsw (block device switch table) 2-12
B_DONE 3-5
B_ERROR

B_ERROR *(continued)*
  described 3-5
  example 3-18, B-10
B_FLUSH flag A-4
B_FORMAT flag A-4
Binaries A-2
Binary distribution 6-11
Bit-mapped graphics 5-3
bkckrecalset B-22
blckbreakup call B-23
blck.c 2-43
blckclose B-9
blck command call B-12, B-13, B-14, B-15, B-17, B-18, B-28, B-19
blck_dma call B-18
blck_doIO call B-17, B-29, B-18
blckDSset call B-19, B-22
blck.h 2-43
blckhalt B-29
blckinit B-8
blckintr
  call B-14
  declared B-8, B-13
blckioctl B-9
blckioset call B-19, B-22
blckISset call B-19, B-22
blcklocal call B-13, B-17, B-28
blckmap call B-11, B-13, B-28, B-16
blckopen call B-30, B-8
blckprint B-30
blckread B-23
blckrecalset call B-19
blckreset call B-12, B-16, B-28, B-25
blckseekset call B-19, B-22
blckspecset call B-19, B-23
blckstart call B-12, B-16, B-27, B-29, B-30, B-11
blckstrategy call B-23, B-10
blcktab B-6
blck_timeout call B-11, B-13, B-27
blckxfer call B-15, B-17
BLKDEV
  blckclose B-9
  blckopen B-8, B-6
Block device
  debugging 6-7
  explained 2-7, 3-1
  I/O 3-1
Block device driver
  example 3-14
  explained 2-4, 2-8
  routines 2-36, 3-4, 3-7
Block on input 8-15

B_NOCROSS flag A-4
Boot prompt 6-6
Bootable rootable 7-18
Booting the new kernel 6-6
B_PHYS 3-5
B_PRIVLG flag A-4
Break processing 8-15
Breakup routine (xxbreakup) 3-13
B_REMOTE flag A-4
B_S52K flag A-4
BSIZE 3-1
BSIZE system block size 2-8
buf structure 2-8, 2-18
  av_forw
    example 3-22
  b_addr 3-9
  b_bcount 3-5
  b_blkno 3-5, 3-9
    example 3-18, B-10
    System V changes A-4, A-16
  b_count
    example 3-18, B-10
  b_cylin
    System V changes A-4
  b_cylin field A-9
  b_dev 3-5, 6-12
    example 3-20
  b_error 3-5
    example B-10
  b_flags
    described 3-5
    example B-10
    System V changes A-4
    test for busy A-8
  b_paddr
    System V changes A-5
  b_proc A-6
  b_reltime A-6
  b_resid
    described 3-5
    example 3-18, B-10
  b_sector
    conversion from b_cylin A-4
    System V changes A-5
  b_sector field A-9
  b_start A-5
  buf_t A-6
  b_un.b_addr A-5
  b_un.b_daddr A-5
  b_un.b_words A-5
  b_want A-6
  read-only members 2-40
  use in block devices 3-5
bufend field A-7
Buffer cache 2-8
Buffers 3-1
bufptr field A-7
buf_t structure A-6
BUSY 8-9
Busy-Wait loops 6-9
B_VERIFY flag A-4
B_WANTED flag A-6

C

Canonical queue 4-10
canput 9-8
Cards 6-12
CARR_ON 4-35, 8-9
CB B-2, B-21, B-25
CBITS5 4-22
CBITS6 4-22
CBITS7 4-22
CBITS8 4-22
cblock 4-8
  System V changes A-6
CBREAK 4-22
CCB 7-23
ccblock structure 4-11
cc(CP)
  driver use 6-2
c_count, c_ptr, c_size fields of ccblock 4-11
CDB 7-4, 7-10
cdevsw 2-12, 9-4
CD-ROM 7-3
CDTR 4-22
CE_ERROR B-5
CEVEN 4-22
Changes
  buf structure A-4
  device number A-3
  kernel routines A-8
  structure definitions A-4
Channels (DMA) 2-28
Character blocks 4-8
Character control block
  data structure 4-11
Character control blocks
  interrupt level control 4-10
Character device 2-7
Character device drivers 2-7, 2-17, 4-1
Character device driver routines 4-2
  interrupt 4-8
Character driver code example 4-14
Character driver routines 2-36
Character expansion and echo 4-12

Character insertion 5-4
Character interface to block devices 3-2
Character list and character block architecture 4-8
Character lists 4-2, 4-8
CHRDEV
  blckclose B-9
  blckopen B-8, B-6
CIENABL 4-15
CINIT 4-15
clear routine
  video driver 5-17
CLESC 8-9
clist
  access in an interrupt routine 4-8
  described 4-2, 4-8
  use in I/O 4-2
clist management 2-36
clists 2-17, 2-39, 2-42
Clock 4-7
CLONEOPEN 9-7
  example 9-11
Close routine
  block driver 3-9
  character driver 4-4
  line printer driver 4-17
  STREAMS driver 9-7
  terminal driver 4-27
cmdresult B-21
cmd_status call B-13, B-14, B-24
cmos routine
  video driver 5-15
CNTRL_WR
  call B-25, B-26, B-27, B-29, B-5
COFF binaries A-2
Collision, vector 6-9
Command codes B-3
Command descriptor block 7-4
Compiler directives A-15
Compiling a driver 6-2
Configurable port addresses 6-13
Configuration changes 6-12
Configuration files A-20
Configuration table 7-6
configure(ADM)
  execution 6-4
  -h option 6-1
  interrupts 6-4
  -T option 2-42
configure(ADM) utility A-19
CONS_GET 5-20
Console driver 5-1
Console messages 2-37
Context switching 2-14
Contiguous memory 2-19
Control register 4-23
Controllers 4-5
  8259 6-9
  debugging 6-7
  line discipline use 8-1
Converting IDDs A-17
Converting installation scripts A-19
Cooked data 4-10
Copy data between kernel addresses 2-37
Copy kernel data to user space 2-37
copy routine
  video driver 5-17
Copy user data to kernel space 2-37
CPARITY 4-22
cram.h header file A-14
CREAD 3-15
CRESET 3-15
Critical code section 2-10
CRTS 4-22
CSTOP2 4-22
Cursor, hardware and software 5-4
custom(C) 6-13
cvttoaddr routine A-12
cvttoint kernel routine A-12
CWRITE 3-15
cyloff 3-15

D

DA_BUSY B-2
Data rates 4-24
db_ fields (struct datab) 9-3
DB_BUSY B-2
DC_BUSY B-2
DD_BUSY B-2
DEBUG ifdef 6-8
Debugging 6-6
Defining registers 4-23
Delay timing 8-13
Delays 2-37
DELETE character 4-10
devbuf structure 2-29, A-7
Device 6-11
Device driver 1-1
  block devices 3-1
  character devices 4-1, 4-10, 4-12
  character interface 3-2
  debugging 6-6
  guidelines 2-39
  interrupt routines 3-22, 4-31, 4-32
  I/O control 4-36

Device driver 1-1 *(continued)*
  line printers 4-13, 4-15
  magnetic tape 4-13
  Modem routines 4-30
  terminals 4-10, 4-21
Device driver routines
  block devices 3-4
  line printer 4-17
Device node, Create A-21
Device number 2-11, A-3
Device register
  line printer 4-16
dev_name 7-7
devnum 7-7
devque B-30
dev_t A-3
DINKERNEL 6-2
DIO B-2, B-24
Direct memory access (DMA) 2-37
  kernel routines 2-28
Dirty buffer 3-1
Disk sort routine 3-6
disk.c disk driver A-16
diskinfo structure 7-6
disksort A-9
DMA
  channels 2-28
  controller 2-28
  dma_alloc routine 2-31
  DMA_BLOCK example 2-32
  dma_enable routine 2-32
  dma_param routine 2-31
  dma_relse routine 2-32
  dmareq example 2-34
  dmareq structure 2-33
  dma_resid routine 2-32
  dma_start routine 2-33
  DMA_Wrmode example 2-32
  d_proc routine 2-31
  driver use 2-28
  example 2-32
  managed transfer 2-31
  memory allocation 2-28
  procedure 2-31
  queued transfer 2-31
  routines 2-30
DMA_CH2 B-18
dma_q B-5
DMA_Rdmode
  example B-18
dmareq structure fields B-18
DMA_Wrmode
  example B-18
d_proc routine 2-31, 2-33
drive macro 3-15
Driver design 2-2
  guidelines 2-39
Driver development package 1-5
Driver installation 6-5
Driver routines
  block routines summary 2-4
  character routines summary 2-5, 4-2
  described 2-3
  param
    example 4-28
  rint
    example 4-33
  SCSI device driver summary 2-6
  SCSI host adapter driver summary 2-6
  video routines summary 2-5, 2-7
  xint
    example 4-32
  xxadapctl 5-3, 5-5, 5-21
  xxbreakup 3-13
    floppy disk driver B-23
    SCSI driver 7-13, A-8
  xxclear 5-3, 5-17
  xxclose
    block driver 3-9
    character driver 4-4
    floppy disk driver B-9
    line printer driver 4-17
    qinit reference 9-4
    STREAMS driver 9-7, 9-12
    terminal driver 4-27, A-9
  xxcmos 5-3, 5-15
  xxcopy 5-3, 5-5, 5-17
  xxd_proc 2-31, 2-33
    floppy disk driver B-18
  xx_entry
    SCSI driver 7-11, 7-20
  xxhalt 4-5
    floppy disk driver B-29, A-14
  xxinit 2-19, 5-3, 5-16
    block driver 3-7
    character driver 4-3
    floppy disk driver B-8
    SCSI driver 7-20
  xxinitscreen 5-3, 5-16
  xxintr 2-15, 2-17, 2-39, 5-7, 8-17
    block driver 3-10
    character driver 4-5
    debugging 6-9
    example in block driver 3-22
    floppy disk driver B-13
    line printer driver 4-19
    SCSI driver 7-11, 7-17, 7-21
    terminal driver 4-31

Driver routines *(continued)*
  xxioctl 5-3, 5-20
    block driver raw I/O 3-12
    character driver 4-7
    floppy disk driver B-9
    SCSI driver 7-13, 7-20
    STREAMS driver 9-13
    terminal driver 4-36, A-9
  xxopen 2-12
    block driver 3-8
    character driver 4-3
    floppy disk driver B-8
    line printer driver 4-17
    qinit reference 9-4
    SCSI driver 7-11, 7-12
    STREAMS driver 9-6, 9-11
    terminal driver 4-25, A-9
  xxpchar 5-3, 5-4, 5-5, 5-18
  xxpoll 4-7
  xxprint 3-10
    floppy disk driver B-30, A-15
  xxproc 4-7, 4-12, 8-10, 8-12
    example 4-36, 8-14
    example call 4-34
    setting t_proc example 4-25
  xxput
    defined 9-5
    qinit reference 9-4
  xxread
    block driver example 3-23
    block driver raw I/O 3-11
    character driver 4-6
    floppy disk driver B-23
    SCSI driver 7-11, 7-14
    terminal driver 4-28, A-9
  xxscroll 5-3, 5-17
  xxscurs 5-3, 5-4, 5-5, 5-18
  xxsgr 5-3, 5-19
  xxsrv
    defined 9-6
    qinit reference 9-4
    STREAMS driver 9-12
  xxstart
    block driver 3-7
      example 3-20
    character driver 4-5
    floppy disk driver B-11
    line printer driver 4-19
    SCSI driver 7-16
  xxstrategy 3-9
    example 3-18
    floppy disk driver B-10
    SCSI driver 7-11, 7-15, A-8

Driver routines *(continued)*
  xxwrite
    block driver example 3-24
    block driver raw I/O 3-11
    character driver 4-6
    floppy disk driver B-23
    line printer driver 4-18
    SCSI driver 7-11, 7-14
    terminal driver 4-28, A-9
DSBIT B-6
dscraddr kernel routine A-12
dscralloc kernel routine A-12
dscrfree kernel routine A-12
d_str 9-4
Dynamic memory allocation 2-20

E

ECHO
  example 4-25
Echoing characters 4-12
Edge-Triggered mode 6-10
EFAULT 8-17, B-9
EINVAL 7-13, 8-17, B-9
EMS 4-23
ENODEV 6-7, 7-12, 7-13, 7-14
Entry routine 7-20
ENXIO 7-12, 7-15, 9-11, B-10
ERASE character 4-10
ERECV 4-23
ERRLIM 3-23
event.c 2-43
ev_exbusmouse.c 2-43
Example block driver code 3-14
exbmouse.c 2-43
Exclusive access
  block driver 3-8
  character driver 4-3
EXEC B-3
EXMIT 4-23
exst 9-10
exst.c 2-43
exstclose 9-12
EXSTFAIL 9-10
  used 9-11
exstinfo 9-10
exstioctl 9-13
exst_lo 9-11
exstm_info 9-9
exstopen 9-11
EXSTOPEN 9-10
  used 9-11

exst_rdq 9-10
exstrinit 9-10
exstsrv 9-12
exst_state 9-10
  tested 9-11
exstwinit 9-10
EXSTWOFF 9-10
EXTPROC 8-9
Extracting files A-19
exvd.c 2-43

F

faddr_t type specification A-14
FALSE B-5
far keyword A-14
FDC_ENB B-2
file.h header file 8-16
FIORDCHK 8-17
fixperm(ADM) 6-14
FL ioctl calls B-7
flbopened
  blckclose B-9
  blckopen B-8, B-6
flbtos B-6
FLBUSY B-7
FL_CHANGE B-24, B-5
fl_cmd B-3
fl_cmds
  values assigned B-22, B-6
flcopened
  blckclose B-9
  blckopen B-8, B-6
fl_dtl B-7
FL_ENB_DI B-2
fl_eot B-5
flerreset B-5
FL_ERROR B-13, B-14, B-15, B-17, B-18, B-2, B-20, B-21, B-24, B-25, B-28, B-29
fl_filf B-7
fl_fmtbuf B-7
fl_gpl B-7
fl_gplf B-7
flhead B-6
FLILLTN B-7
flinitflg B-5
FLIOCSIZE B-9
FLIOERR B-7
fl_job B-19, B-27, B-5
fllocalbuf B-6
fllocaltransfer B-6
fl_lsn B-5
fl_ma B-5
fl_moft B-7
fl_mont B-7
fl_name B-8
fl_nbs B-7
FLNOTDS B-7
FLNOWR B-7
Floating-point arithmetic 2-39
fl_onecyl B-6
flopen B-30
Floppy disk driver example B-1
Floppy drive command codes B-3
Floppy table B-3
flparam B-8
flptab
  default parameters B-8
  structure definition B-7
flresidcnt B-5
FLSECSZ B-5
FLSHIFT B-5
FLSMASK B-5
fl_sn B-5
fl_spc1 B-7
fl_spc2 B-7
flstate B-5
fl_stats B-6
flstob B-6
fltmpaddr B-6
fl_tn B-5
fl_to B-5
fltranscnt B-5
fltrnsferred B-5
Flush output queue 8-13
flushq 9-8
flushtlb 2-37
fm_hd B-7
fm_il B-7
fm_sec B-7
fm_size B-7
fm_status B-7
fmtfl B-7
fm_trk B-7
FNDELAY, FREAD, FWRITE, FSTOPIO 8-16
FORMAT
  default value B-3
  defined B-3
Frames (memory) 2-15
FREAD, FWRITE, F flags
  block driver 3-8
  character driver 4-3
freeb 9-8
freemsg 9-8
ftoof kernel routine A-12
ftoseg kernel routine A-12

G

Get driver parameters 6-7
getc 4-8
getcbc kernel routine A-12
getq 9-8
Group codes (SCSI) 7-4
Guidelines for driver writing 2-39

H

Half-Duplex pipe 9-8
Halt routine (xxhalt) 4-5
Handshake routines B-24
ha_num 7-7
Hard disk driver 3-14
Hard disk routines
  hdintr 3-22
  hdread 3-23
  hdstart 3-20
  hdstrategy 3-18
  hdwrite 3-24
Hardware cursor 5-4
hdinstall command (obsolete) A-21
hdintr routine 3-22
hdread routine 3-23
hd_sizes 3-17
hdstart routine 3-20
hdstrategy routine 3-18
hdwrite routine 3-24
head, STREAMS 9-2
Header files
  buf.h 2-18
  iobuf.h 2-18
  naming convention 1-4, A-4
High-water mark 4-9, 8-13
  line printer definition 4-16
HIWAT 4-16
Host adapter 7-1
Host adapter driver 7-20
Host adapter drivers 2-6

I

IASLP 8-9
ICANON 8-17
  example 4-25
id (SCSI) 7-7
IDDs A-17
idscsi 7-6, 7-18
I_ERRNAK 9-15
I_ERROR 9-15
ifdef DEBUG 6-8
ifdef M_S_UNIX A-15
IGNBRK 4-34
I_INTARG 9-14
IMS 4-23
in kernel routine A-13
Include files A-14
init routine
  block driver 3-7
  video driver 5-16
Initialization routine
  memory allocation 2-19
  SCSI host adapter driver 7-20
  video driver 5-7
  xxinit 4-3
initscreen routine
  video driver 5-16
I_NOARG 9-14
Insert a character 5-4
Install kernel A-21
Installable device driver (IDD) A-17
Installation
  video driver 5-6
Installation script 6-15
Installation scripts A-19
Installing a driver 6-5
Installing a SCSI driver 7-18
Intelligent controller example driver 3-14
International character set mapping 2-37
Interrupt controller chip 6-10
Interrupt enable register 4-23
Interrupt explained 2-13
Interrupt handler configuration 6-4
Interrupt identification register 4-24
Interrupt level control
  character control blocks 4-10
Interrupt routine 2-18
  block driver 3-10
    example 3-22
  character driver 4-5
  line printer driver 4-19
  SCSI driver 7-17
  SCSI host adapter driver 7-21

Interrupt routine 2-18 *(continued)*
   terminal driver 4-31
   terminal receive interrupt 4-33
   terminal transmit interrupt 4-32
   video driver 5-7
Interrupt service routines 2-15
Interrupt time processing 2-16
Interrupt vector conflicts A-20
Interrupt vectors 2-42, 6-3, 6-12
   collision 6-9, A-16
   philosophy 6-12
Interrupts
   poll first 6-8
Invoke the new kernel A-21
I/O port input 2-37
I/O port output 2-37
I/O ports 2-40, 6-7
iobuf structure 3-6, 6-4
IOC_ ttiocom(K) conditions 8-17
ioctl routine
   block driver 3-12
   character driver 4-7
   SCSI driver 7-13
   SCSI host adapter driver 7-20
   video driver 5-20
ioctl(S) 4-7
IOshake call B-20, B-25
IRECV 4-23
IRQ bus leads 6-9
IRQ (interrupt request) A-16
IS386 kernel routine A-13
I_SETERR 9-15
I_SETHANG 9-15
ISOPEN 4-35, 8-9
IXMIT 4-23

K

kbgrp structure 5-8
Kernel routines
   btoc 2-19
   canon 4-10, 8-12
   cmn_err 3-10
      debugging use 6-7
      example A-15, B-17, B-27, B-30, A-10
   copyin 4-6
   copyio
      example B-28, B-29
   copyout 4-6
      example B-9
   cpass 4-6
      example 4-18

Kernel routines *(continued)*
   ctob 2-19
   db_alloc 2-28, A-7
   db_free 2-28
   db_read 2-29, 4-6
   db_write 2-29, 4-6
   delay 3-7
   described 2-3
   deverr
      example 3-22
   disksort 3-6, 3-9
      example 3-18, B-30
      SCSI driver 7-15, A-9
   dma_alloc 2-31
      example B-30
   dma_breakup 3-13
      example B-23, A-8
   dma_enable 2-31, 2-32
      example B-18
   dma_param 2-31
      example B-18
   dma_relse 2-31, 2-32
      example B-11
   dma_resid 2-32
   dma_start 2-33
   flushtlb 2-37, 2-40
   fubyte 4-6
   fuword 4-6
   getc 4-8
      example 4-19
   getchar
      debug use 6-8
   geteblk 3-7
   inb
      caveat 6-7
      example 3-22, B-21, B-24
   iodone
      example 3-18, 3-22, B-10, B-16,
         B-27, B-29
   iowait 3-7
   ktop
      example B-29, A-16, A-5
   longjmp 3-7
   major A-9
   memget 2-19
   minor
      example 3-15, A-15, A-9
   minor(K) 2-12
   obsolete routines A-12
   outb
      debugging 6-7
      example 3-20, B-20
   paddr
      example 3-20, B-11, A-5

Kernel routines *(continued)*
- physck 3-7
  - example B-23, A-8
- physio
  - example 3-23, B-23
  - use in raw I/O 3-3, A-8
- pio_breakup 3-13, A-8
- printcfg 3-7, 4-3
  - example B-8
  - initialization messages 6-11
- printf
  - debugging use 6-7
- ptok A-16
- putc 4-8
  - example 4-18
- scsi_getdev 7-13
- seterror
  - example 4-25
- signal
  - example 4-35
- sleep 2-23, 3-7, 4-3, 8-17
  - example 4-18
  - ps(C) interaction 6-10
  - PZERO usage 4-16
- sleep(K) 2-14, 2-16, 2-17
- spl5
  - devices 6-9
  - example B-27
- spl6 4-7
- spl7
  - debugging use 6-9
  - example 8-14
  - line discipline use 8-14
- splbuf
  - example 3-18
- splcli
  - example 4-18
- spl(K) 2-17, 2-39
- splx
  - debugging use 6-9
  - example 4-18
- sptalloc 2-20, 2-24
  - video memory management 5-6, A-11
- sptfree 2-23, 3-7, A-11, A-13
- subyte 4-6
- suword 4-6
- timeout 3-7, 4-3
  - debugging use 6-10
  - example 4-38, 8-15, B-11, B-12, B-13, B-14, B-27
- ttclose 8-6
- ttin 8-6
- ttinit 4-26
  - example 4-25

Kernel routines *(continued)*
- ttiocom 8-12, 8-16
  - conditions 8-17
  - example 4-36
- ttioctl 8-6
- ttopen 8-6, 8-11
- ttout 8-6, 8-10
- ttread 4-6, 8-6
- ttrstrt 8-12
  - example 4-38, 8-15
- tttimeo 8-9
- ttwrite 4-6, 8-6
- ttxput 8-9
- tty 8-12
- ttyflush 8-10, 8-12
  - example 4-35
- ttywait 8-10
- vas 2-25
  - video memory management 5-6
- vasbind 2-21, 2-25
- vasmalloc 2-25
- vasmapped 2-26
- vasunbind 2-26
- video 2-43
- vtop A-5
  - example B-11
- wakeup 4-8
  - example 4-19
- wakeup(K) 2-14, 2-17
- xxinit restrictions
  - block driver 3-7
  - character driver 4-3

Kernel stack 2-15
kernel/space.c file 8-3
kg_* (struct kbgrp) 5-13
Kill character 4-10
KVBASE 2-26

L

layers(C) 4-4
L_BREAK 4-34
l_close routine 4-27, 8-3
ldtalloc kernel routine A-13
ldtfree kernel routine A-13
len 3-15
Line discipline 9-1
- creating one 8-6
- explained 8-1
- one 8-2
- two 8-2
- zero 8-2

Line printer routines
  device driver 4-10
  interrupt routines 4-19
  lpclose 4-17
  lpintr 4-19
  lpopen 4-17
  lpstart 4-19
  lpwrite 4-18
Line printers 4-13
  device driver 4-15
linecnt variable 8-3
linesw structure definition 8-3
Link kit 6-4
Linked list (STREAMS) 9-2
link_id and link_ptr 7-8
Linking a driver 6-5
link_unix 6-5
link_unix edit A-20
l_input routine 4-11, 4-34, 8-3
l_ioctl routine 8-3
l_mdmint routine 8-3
Logical unit number 7-4
Logical vector number 6-3
Loop back driver 9-8
l_open routine 4-26, 8-3
l_output routine 4-11, 4-37, 8-3, 8-14
LOWAT 4-16
Low-water mark 4-9
  line printer definition 4-16
lpclose routine 4-17
lp_flags 4-16
lpintr routine 4-19
lpopen routine 4-17
LPPRI sleep(K) priority 4-16
lp_queue 4-16
lpstart routine 4-19
lpwrite routine 4-18
l_read routine 4-28, 8-3
LUN 7-4
l_write routine 4-28, 8-3

M

m6845.c 2-43
m6845.h 2-43
machdep.h header file A-15
Magnetic tape drives 4-13
Mailboxes 7-23
Major device number 6-4
Major number 2-11
  use 6-12, A-20
major(K) A-3
MAPCONS 5-20
mapphys kernel routine A-13
mapptov kernel routine A-13
masm(CP) 6-2
master file A-17
Master request line 6-3
MAX_RD B-3
MAXRETRY B-5
maxvec A-16
MAX_WR B-3
mb_* (struct mscrn) 5-9
M_CTL 9-13
M_DATA 9-13
mdevice file 6-1
  use with xxinit routine
    block driver 3-7
    character driver 4-3, A-17
memget routine 2-19
Memory allocation
  contiguous 2-19
  DMA use 2-28
  dynamic 2-20
  memory mapped I/O 2-24
  physical 2-19
  shared memory 2-25
Memory management 2-19, 2-37
  video driver 5-6
Memory-mapped I/O 2-24
Message block 9-3
Message display 2-37
MF B-3
  used B-22
mf_* (struct mscrn) 5-9
M_FLUSH 9-13
Minor number 2-11
minor(K) A-3
M_IOCTL 9-12
mk_* (struct mscrn) 5-9
mknod(C) command A-21
Modem interrupts 4-35
Modem routines 4-30
Modes of operation 2-13
MODOPEN 9-7
module, STREAMS 9-2
module_info 9-4
motor, turn off B-27
motoron call B-12, B-14, B-26
M_PCPROTO 9-13
M_PROTO 9-13
mscrn
  described 5-8
  insert a character 5-4
  structure access 5-7
  switch multiscreens 5-5

mscrn structure
  relation to multiscreen number 5-3
mscsi file 7-19
msgb structure 9-3
msp
  switch multiscreens 5-5
mspu
  relation to multiscreen number 5-3
M_S_UNIX definition A-15
MT
  defined B-3
mtune file A-17
Multiple initiators 7-3
Multiscreen
  insert a character 5-4
Multiscreen number 5-3
Multiscreen switch 5-5
mv_* (struct mscrn) 5-9
mv_col
  software cursor 5-4
mv_row
  software cursor 5-4
mv_savscrn 5-9
mydebugflg 6-8

N

Naming a driver 6-11
Naming conventions 1-4
NBPC 3-16
NBPS 3-16
NCMDS B-3
NCPD 3-16
Near and far keywords A-14
near keyword A-14
NEXST 9-10
NHD 3-16
No delay open
  block driver 3-8
  character driver 4-3
No such device 6-7
NON_DMA B-2
Non-text mode 5-3
NPARTS 3-16
NSPC 3-15
NSPT 3-16
nswap A-16
NTDEVS 4-23
NTPC 3-16
nulldev 8-3, 8-6
NUM_RD B-3
NUM_WR B-3

O

OASLP 8-9
Obsolete kernel routines A-12
ONLCR 4-25
Open device as swap device 4-4
Open flags
  character driver 4-3
Open routine
  block driver 3-8
  character driver 4-3
  line printer driver 4-17
  SCSI driver 7-12
  STREAMS driver 9-6
  terminal driver 4-25
OPENFAIL 9-7
  example 9-11
open(S) relationship to xxioctl 3-12
open(S) system call 2-12
Operation modes
  kernel, system, or user mode 2-13
OPOST 4-25
OTYP_ identification values
  block driver 3-8
  character driver 4-4
out kernel routine A-13
Output queue 4-10

P

Packaging 6-15
paddr(K) routine A-5
Page present 2-22
Pages, memory 2-21
Panic the system 2-38
param routine 4-28, 8-15
Parameter passing to device drivers 2-41
parser 5-3
part macro 3-15
partab structure 3-15
pchar routine
  video driver 5-18
Peripheral controllers 6-7
Permissions (check for super user) 2-38
permlist 6-13
  sample 6-14
PERROR 4-34
P_FLCT B-2
P_FLCTL B-2
  CNTRL_WR call B-25, B-26, B-27, B-29
  in a marcro B-5

P_FLDCR B-2
  read B-20, B-21
  write B-20
P_FLDMA B-2
P_FLMSR B-2
  read B-21, B-24, B-25
P_FLTC B-2
PG_P, PG_RW, PG_US 2-22
Physical address A-5
Physical I/O 2-28, 6-12, 7-14
  described 3-2
Physical memory allocation 2-19
physio(K)
  buffers A-8
  use in raw I/O 3-3
PIC (programmable interrupt controller) 2-13
Pipe, STREAMS 9-8
Poll routine (xxpoll) 4-7
Pollable state information 6-10
Polling 6-8
Port addresses 6-7
Port addressing 6-13
POSIX ttiocom(K) extension 8-18
Possible problem areas A-16
Preconfigured drivers 6-11
Prefix naming 6-11
Prefix selection 6-1
Primary card 5-15
Print routine (xxprint) 3-10
Problem areas A-16
Proc routine
  described 8-12
  example 4-36
Proc routine (xxproc) 4-7
proc structure 2-40
Processes
  system 2-13
  u-area 2-14
  user 2-13
Programmable interrupt controllers 6-9
Programmed I/O 2-38
ps(C) 6-10
put routine 9-3
Put routine 9-5
putbq 9-8
putc 4-8
putcbc kernel routine A-13
putctl 9-8
putq 9-8
PZERO
  example 2-34, 4-15

Q

qinit 9-4
  default values 9-10
qreply 9-8
Quad-word boundary 6-2
queue, STREAMS 9-2
queue_t structure 9-2

R

RADDRH 3-15
RADDRL 3-15
Raw input data vs. cooked data 4-10
Raw input queue 4-10
Raw I/O 2-8, 3-2
  block driver routines 3-11
  character interface 4-13
  open device 3-8
RBASE
  disk driver example 3-16
  line printer driver register 4-16
rblckbuf B-6
RCMD 3-15
RCMD register 3-15
RCNT 3-15
RCNTRL 4-16, 4-23
RCYL 3-15
RD 9-8
RDATA 4-16
RD_DATA 8-3
  condition B-19
  default value B-3
  used B-18, B-22
RD_DEL_DATA B-3
  default value B-3
RD_ID B-3
  default value B-3
RDshake call B-20, B-21, B-25
RD_TRK B-3
  default value B-3
Read routine
  block driver 3-11
    example 3-23
  character driver 4-6
  SCSI driver 7-14
  terminal driver 4-28
RECAL B-3
  condition B-19
  default value B-3
  used B-12, B-14, B-22

recalstat B-5
Receive interrupt routine 4-33
Re-entrant processing 2-10
Register
  control 4-23
  interrupt enable 4-23
  interrupt identification 4-24
Register handshake routines B-24
Registers 6-7
  defining 4-23
Relink the kernel A-21
REQ_IO structure 7-8
REQ_SEND 7-8
Request block 7-8
Request line 6-3
Required video routines 5-3
Resume terminal output 8-14
RIENABL 4-23
RIIR 4-23
rint routine 4-33
Rising edge 6-10
Root device 6-12
RQM B-2, B-24, B-25
RRDATA 4-23
RSEC 3-15
RSPEED 4-23, 4-25
RSTAT 3-15
RSTATUS 4-16, 4-23
RTDATA 4-23
RTO 8-9
RTRK 3-15

S

s5param.h
  BSIZE reference 3-1
Sample drivers 2-43
Sample installation script 6-15
Save area 5-5
sc_hd B-7
sc_nb B-7
Screen save area 5-5
scroll routine
  video driver 5-17
sc_sec B-7
SCSI
  ANSI standard 7-2
  bus device ID number 7-19
  bus device LUN 7-19
  CCB 7-23
  CDB 7-4, 7-10
  commands 7-4
SCSI *(continued)*
  communication on bus 7-3
  configuration table 7-6
  device configurations 7-3
  device driver
    breakup routine 7-13
    init routine not needed 7-11
    installation 7-18
    interrupt routine 7-11, 7-17
    ioctl routine 7-13
    open routine 7-11, 7-12
    read routine 7-11, 7-14
    start routine 7-16
    strategy routine 7-11, 7-15
    write routine 7-11, 7-14
  devices 7-3
  diskinfo structure 7-6
  driver names 7-18
  group codes 7-4
  host adapter
    communication with a driver 7-5
    described 7-20
    entry routine 7-7, 7-11, 7-20
    explained 7-1
    init routine 7-20
    interrupt routine 7-21
    ioctl routine 7-20
    number 7-19
    routine summary 2-6
  host adapter driver
    installation 7-18
  idscsi 7-18
  initiators 7-3
  installing a driver 7-18
  io_intr routine 7-10, 7-21
  kernel routines 2-38
  link_id and link_ptr 7-8
  local structures 7-6
  maximum number of devices 7-3
  mscsi file 7-19
  REQ_SEND 7-8
  request block 7-8
  routine summary 2-6
  SCSI bus explained 7-1
  SCSI bus speed 7-3
  SCSI described 7-1
  scsi_cdb 7-10
  scsi_dev_cfg structure 7-7
  scsi_getdev 7-13
  scsi_io_req 7-8, 7-21
  structures 7-6
  xx_entry routine 7-11
sctrhdr B-7
sc_trk B-7

SCTS 4-22
scurs routine
  video driver 5-18
sdevice file 6-3
SDRV_RDY B-2
SDSR 4-22
secbits B-6
Security feature
  use of FSTOPIO
    block driver 3-8
    character driver 4-4
SEEK
  condition B-19
  default value B-3
  defined B-3
  used B-12, B-15, B-22, B-28
select(S) access 2-38
SELREAD and SELWRITE 8-17
SENS_DR
  condition B-19
  default value B-3
  defined B-3
  used B-17, B-22
SENS_INTR
  condition B-19
  default value B-3
  defined B-3
  used B-14, B-22
Serial console 5-16
Serial device interrupts 6-10
Serial drivers 6-8
Service routine 9-6, 9-12
Set priority level (spl) 2-38
setjmp kernel routine A-13
SFERR 4-22
sgr routine
  video driver 5-19
SGR_* values 5-19
Shared interrupt vectors 6-9
Sharing interrupt vectors 2-42
shl(C) 8-2, 8-13
Shutdown 6-6
shutdown(ADM) A-21
sidd 6-13
SIGHUP 4-35
Single initiator 7-3
sizbits B-6
SIZEBITS B-6
SK
  defined B-3
  used B-22
Slave chip interrupts A-16
Slave controller 6-3
Small computer systems interface (SCSI) 2-38
Smart controllers 8-2
SOERR 4-22
Software cursor 5-4
Sorting I/O requests 3-6
Special device files 2-11
SPECIFY
  condition B-19
  default value B-3
  defined B-3
  used B-13, B-14, B-23
SPERR 4-22
sptalloc(K) routine 2-20, 2-24
sptfree(K) routine 2-23
sptmap 2-21
Spurious interrupt 4-20
SRRDY 4-22
srtest.c 2-43
srv routine 9-12
Stack
  local 2-39
  u-area 2-14
Start routine
  block driver 3-7
    example 3-20
  character driver 4-5
  line printer driver 4-19
  SCSI driver 7-16
Static variables 2-17
Status register 6-7
STFMSEEK B-5
STIO B-5
STMOTOR B-14, B-5
Strategy routine 7-15
Strategy routine (xxstrategy) 3-9
STRDY 4-22
STREAMS
  driver 9-2
  head 9-2
  module 9-2
  overview 9-1
  queue 9-2
STREAMS driver 9-8
STREAMS service routine 9-12
streamtab 9-10
STRECAL B-15, B-5
STRESET B-14, B-5
strioctl 9-18
strread 9-18
STRSEEK B-15
Structure header files A-4
strwrite 9-17
STSEEK B-5
Style issues for user prompts 6-11
Suspend output 8-14

Swap device 6-12
Swap device open 4-4
Switch layers 8-13
Switch multiscreens 5-5
SWR_PROT B-2
Synchronous writes
  block driver 3-8
  character driver 4-4
System clock 4-7
System mode 2-13, 2-18
System mode stack 2-14
System processes 2-13

T

T_ conditions in xxproc 8-12
t_ members of the tty structure 8-7
TACT 8-9
Task-time
  character driver 4-2
Task-time processing 2-16
TBLOCK 4-38, 8-10
T_BLOCK
  example 4-38
T_BLOCK xxproc condition 8-10
T_BREAK 4-38
TC ttiocom(K) conditions 8-17
t_canq 4-10, 8-8
  example 4-26
t_cc 8-9
t_cflag 4-26, 8-8
TCGETA 8-8
TCSETA 8-8
TCXONC 8-17
tdclose routine 4-27
tdintr routine 4-31
tdioctl routine 4-36
tdmint routine 4-35
tdmodem routine 4-30
tdopen routine 4-25
tdparam 4-26
tdparam routine 4-28
tdproc routine 4-36
tdread routine 4-28
tdrint routine 4-33
td_speeds 4-24
tdwrite routine 4-28
tdxint routine 4-32
Terminal
  block on input 8-15
  break processing 8-15
  canonical queue 4-10
  data speeds 4-24
  device driver sample 4-21
  flush read queue 8-14
  flush write queue 8-14
  line discipline 8-1
  output queue 4-10
  raw input queue 4-10
  resume output 8-14
  suspend display 8-14
  timeout 8-14
Terminal device drivers 4-10
Terminal drivers
  tdioctl 4-36
  tdmint 4-35
  tdxint 4-32
Terminal handler 8-1
Terminal routines
  tdclose 4-27
  tdintr 4-31
  tdmodem 4-30
  tdopen 4-25
  tdparam 4-28
  tdproc 4-36
  tdread 4-28
  tdrint 4-33
  tdwrite 4-28
Terminals
  clist use 4-8
Text mode 5-3
  multiscreen switch 5-5
t_iflag 4-25, 8-8
TIMEOUT 8-10, 8-14
Timing 2-38
TLB 2-37
t_lflag 4-25, 8-8
  ttiocom 8-17
t_line 4-26, 8-3, 8-9
t_oflag 4-25, 8-8
T_OUTPUT 4-37
t_outq 4-10, 8-8, 8-10
t_pgrp 8-9
TPI96BIT B-6
t_proc 8-8
  example 4-25
Translate lookaside buffer (TLB) 2-37
Transmit interrupt routine 4-32
t_rawq 4-10, 8-8
t_rbuf 4-11, 8-8
  fields set up 4-34

T_RESUME
  example 4-37
T_RFLUSH
  example 4-38
TRUE B-5
t_state 8-8
  example 4-26
  ORing in values 4-37
T_SUSPEND
  example 4-37
t_tbuf 4-11, 8-8
  field set up 4-37
  flush output queue 8-14
T_TIME
  example 4-37
TTIOW 8-10
TTOPRI 8-10
TTSTOP 4-37, 8-10
  example 8-14
TTXOFF 4-38, 8-10
TTXON 4-38, 8-11
  example 8-15
tty drivers 4-10
TTY routines 2-38
tty structure 2-18, 8-7
TTYHOG 8-14
Tunable kernel parameters A-19
T_UNBLOCK
  example 4-38
T_UNBLOCK xxproc condition 8-11
T_WFLUSH
  example 4-37

U

U-area 2-14, 2-16, 2-18, 2-37, 2-42
UART 4-21
UNITBITS B-6
unitbits macro A-15
unlinkb 9-8
unmapphys kernel routine A-13
User mode 2-13
User processes 2-13
User prompting 6-11
user structure
  read-only members 2-40
  System V changes A-6
  u_base 2-41
  u_count 2-41, A-8
  u_offset 2-41, A-8
  u_segflg 2-41
USERMODE 4-7

V

v_* (struct adapter) 5-11
v_adapctl 5-5
vas(K) routines 2-25
v_curscrn
  msp value stored 5-4
vecintsw table A-16
VECT0 4-23
VECT1 4-23
Vector collision 6-9
Vector number 6-3
Vectors 6-9
Vendor unique (SCSI) 7-4
vid.c console driver 5-1
Video driver
  init routine 5-7
  insert a character 5-4
  interrupt routine 5-7
  memory management 5-6
  required routines 5-3
vidloops.c 2-43
Virtual memory 2-21, 2-26
Virtual to physical address translation A-5
VSWTCH 8-13
VTIME 8-9, 8-13
v_type 5-16

W

wakeup 4-8
WOPEN 8-11
WORM drive 7-3, 7-18
WP_ERROR B-15, B-17, B-2
WR 9-8
WR_DATA B-3
  condition B-19
  default value B-3
  used B-18, B-22
WR_DEL_DATA B-3
  default value B-3
Write protect check B-17
Write routine
  block driver 3-11
    example 3-24
  character driver 4-6
  line printer driver 4-18
  SCSI driver 7-14
  terminal driver 4-28
Writing a line discipline 8-6

WR_PROT B-2
WRshake call B-20, B-24

X

X3.131 ANSI standard 7-2
XCGETA 8-18
XCSETA 8-18
XCSETAF 8-18
XCSETAW 8-18
XENIX files relocated under System V A-17
XENIX vs. System V directory structures A-18
xinit
  video driver 5-7
xint routine 4-32
XOFF 4-35
XON 4-35
x.out binaries A-2
xxadapctl
  described 5-21
  information 5-3
  multiscreen switch 5-5
xxbreakup routine 3-13
  floppy disk driver B-23
  SCSI driver 7-13
xxclear 5-17
  information 5-3
xxclose
  block driver 3-9
  character driver 4-4
  floppy disk driver B-9
  line printer driver 4-17
  STREAMS driver 9-7
  terminal driver 4-27
xxcmos 5-15
  information 5-3
xxcopy 5-17
  character insertion 5-5
  information 5-3
xxd_proc 2-31
  floppy disk driver B-18
xx_entry
  SCSI driver 7-11, 7-20
xxhalt 4-5
  floppy disk driver B-29
xxinit 5-16
  block driver 3-7
  floppy disk driver B-8
  information 5-3
  SCSI driver 7-20
xxinitscreen 5-16
  information 5-3
xxintr
  block driver 3-10
    example 3-22
  character driver 4-5
  floppy disk driver B-13
  line printer driver 4-19
  SCSI driver 7-11, 7-17, 7-21
  terminal driver 4-31
  video driver 5-7
xxioctl 5-20
  block driver raw I/O 3-12
  floppy disk driver B-9
  information 5-3
  SCSI driver 7-13, 7-20
  terminal driver 4-36
xxopen
  block driver 3-8
  character driver 4-3
  floppy disk driver B-8
  line printer driver 4-17
  SCSI driver 7-11, 7-12
  STREAMS driver 9-6
  terminal driver 4-25
xxpchar 5-18
  character insertion 5-5
  information 5-3
  software cursor 5-4
xxpoll 4-7
xxprint 3-10
  floppy disk driver B-30
xxproc 4-7
  called by ttiocom TCXONC condition 8-17
  example 4-36
  section describing 8-12
  T_BLOCK condition 8-10, 8-12
    example 8-15
    ttiocom call 8-17
  T_BREAK condition, 8-12
    example 8-15
  T_DISCONNECT condition 8-12
  T_INPUT condition 8-12
  T_OUTPUT condition 8-12
    example 4-37
    terminal driver call 4-32
  T_PARM condition 8-13
    example 8-15
  T_RESUME condition 8-13
    example 4-37, 8-14
    terminal driver call 4-34
    ttiocom call 8-17
  T_RFLUSH condition 8-13
    example 8-14
  T_SUSPEND condition 8-10 8-13
    example 4-37, 8-14

xxproc 4-7 *(continued)*
T_SUSPEND condition 8-10 8-13 *(continued)*
terminal driver call 4-34
ttiocom call 8-17
T_SWTCH condition 8-13
T_TIME condition 8-13
example 4-37, 8-14
T_UNBLOCK condition 8-11 8-13
example 8-15
ttiocom call 8-17
T_WFLUSH condition 8-13
example 4-37, 8-14
xxput 9-5
xxread
block driver
example 3-23
block driver raw I/O 3-11
character driver 4-6
floppy disk driver B-23
SCSI driver 7-11, 7-14
terminal driver 4-28
xxscroll 5-17
information 5-3
xxscurs 5-18
character insertion 5-5
hardware cursor 5-4
information 5-3
xxsgr 5-19
information 5-3
xxsrv 9-6
xxstart
example call 8-14
SCSI driver 7-16
xxstart routine
block driver 3-7
example 3-20
character driver 4-5
floppy disk driver B-11
line printer driver 4-19
xxstrategy
SCSI driver 7-11, 7-15
xxstrategy routine 3-9
floppy disk driver B-10
xxtab structure 3-6, 6-4, 7-6, 7-17
example
declaration 3-17
field access 3-20
setting fields example B-16
xxwrite
SCSI driver 7-11, 7-14
xxwrite routine
block driver
example 3-24
xxwrite routine *(continued)*
block driver raw I/O 3-11
character driver 4-6
floppy disk driver B-23
line printer driver 4-18
terminal driver 4-28

514-000-051
24124

Hit a "stumbling block"? Tell us where.

Manual title:	SINIX Open Desktop V1.0, U5753-J-Z95-1-7600
Page	Problem:

I am	☐ a programmer	I use the manual	☐ frequently
	☐ a system administrator		☐ occasionally for reference
	☐ an ordinary user		☐ ______________
	☐ ______________		

Manual title:	SINIX Open Desktop V1.0, U5753-J-Z95-1-7600
Page	Problem:

I am	☐ a programmer	I use the manual	☐ frequently
	☐ a system administrator		☐ occasionally for reference
	☐ an ordinary user		☐ ______________
	☐		

Suggestions – Criticisms – Corrections

Are you happy with this manual? If so, let us know.
If not, help us improve it by informing us
- where you have noticed mistakes
- where the content is unclear.

From:

Name ______

Company/department ______

______

Address ______

______

Postal Code ______

______

Telephone ( ) ______

Local Siemens office ______

Contact person ______

Siemens AG
DI ST QM2
Manualredaktion
Otto-Hahn-Ring 6
Postfach 83 09 51

D-8000 München 83

From:

Name ______

Company/department ______

______

Address ______

______

Postal Code ______

______

Telephone ( ) ______

Local Siemens office ______

Contact person ______

Siemens AG
DI ST QM2
Manualredaktion
Otto-Hahn-Ring 6
Postfach 83 09 51

D-8000 München 83