XENIX® System V

Development System

Device Driver Writer's Guide

SCO Document Number: XG-10-31-88-5.0/2.3

Contents

Chapter 1

Introduction to Device Drivers

1.1 Introduction

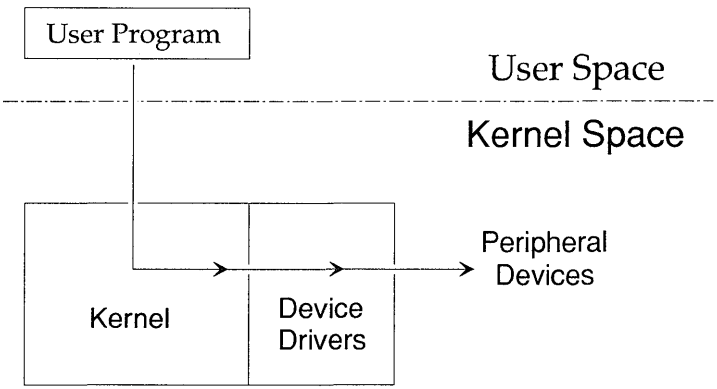
This chapter, along with the "Example Driver Code" chapter, explains how to write and install device drivers in a XENIX environment. It describes the role of device drivers in a XENIX-based system, and discusses other special considerations involved in writing a device driver. It describes the XENIX model of devices in terms of files, tasks to be performed, and interrupts to be processed.

These chapters are meant as a reference for the experienced XENIX C Language developer who wishes to use previously unsupported hardware. Writing a device driver is a complex and difficult task, and should not be undertaken lightly. You must have a technical reference both for the device you wish to support and for the computer that will be using the driver and device.

1.1.1 What is a XENIX Device Driver?

For each peripheral device (such as a terminal) in an SCO XENIX system, there must be a *device driver* to provide the software interface between the device and the system. An SCO XENIX device driver is a set of routines that communicates with a hardware device, and provides a uniform interface to the SCO XENIX kernel. This interface lets the kernel interpret your I/O requests as operating-system tasks to be performed.

Specifically, the XENIX device driver manages the flow of data and control between the user program and the peripheral devices. The path of an I/O request is shown in the following diagram, starting with a system call from a user program, and ending at the device driver:



1.1.2 Device Models Supported by XENIX

The XENIX operating system supports two device models: *character devices* and *block devices*. This chapter describes how to write device drivers for both device models.

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 XENIX 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 communicate directly with the user program. The process 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 require more involvement from the operating system to perform their tasks. 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; magnetic tapes are often read or written using block I/O.) 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 1024 bytes in the 286 and 386 versions of XENIX. 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. XENIX commands the device driver to read and write from the buffers as necessary. It manages

these buffers to perform services such as blocking and unblocking of data and disk caching.

1.1.3 Using Example Driver Code

The "Example Driver Code" chapter discusses sample device-driver source code for a line printer, a terminal, a hard-disk drive, and a memory-mapped screen. These source-code samples are intended as prototypes from which the experienced programmer can begin writing a device driver for a particular device.

1.1.4 About Special-Device Files

To a XENIX 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 8 port serial card have the same major device number, but they would have 8 separate minor device numbers.

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 *XENIX User's Reference*. The file appears in a directory and has owner and permission fields, as does any disk 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:

```
crw--w--w- 1 michaelb    user     5,  6 Sep 21 07:21 /dev/tty1c
crw--w--w- 1 zursch      user     5,  7 Sep 21 09:49 /dev/tty1d
brw------- 1 sysinfo     backup   3,  2 Sep 21 05:34 /dev/hd01
```

Here the */dev/tty1c* file has a major device number of 5 and a minor device number of 6. */dev/tty1d* has a major device number of 5 and a minor device number of 7. The */dev/hd01* file has a major device number of 3 and a minor device number of 2.

When a user process opens the special device file, the XENIX 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 */usr/sys/conf/c.c* file that is generated by the **config**(ADM) program when the kernel is built.

When a user process uses the **open**(S) or **fopen**(S) system service on a desired file, the XENIX kernel calls the device driver's open entry-point through the *cdevsw* or *bdevsw* table, supplying the minor device number as an argument. The minor 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 "use double density" in the case of a floppy disk.

These special device files should have meaningful names and should reside in the */dev* directory. For example, */dev/tty03* would normally be associated with the major device number of the console device driver; its minor number would indicate the fourth port. Note that this is just a convention; the system administrator could assign the same major/minor numbers to either of the files */usr/ellen/magtape* or */usr/ellen/tty91*, with identical results. The name is for your convenience; the XENIX kernel keys solely on the major and minor device numbers.

1.2 Kernel Environment

This section briefly discusses a few functional aspects of the XENIX 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 XENIX 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 XENIX System V kernel depends on interrupts to schedule processing efficiently.

1.2.1 Version 7/System V Compatibility Issues

This section describes some of the changes between Version 7 of UNIX and System V of XENIX that affect the device-driver interface.

Device Numbers

In Version 7 of UNIX, the *dev* parameter passed to the **open**, **close**, **read**, **write**, and **ioctl** driver routines included the major and minor device numbers. In System 3 and System V, only the minor device number is passed in the *dev* parameter. This means it is no longer necessary for all device drivers to mask out the major device number before checking the minor device number.

iomove

Some Version 7 device drivers used a routine called **iomove** to copy to or from user space. The **iomove** routine does not exist in System 3 and System 5. However, adding the following code will provide most of the same capability:

```
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/dir.h"
#include "../h/user.h"

iomove(cp, cnt, flag)
caddr_t cp;
register int cnt;
int flag;
{

    register int dirflag;

    if (cnt == 0)
          return;               /* Nothing to do! */

    dirflag = (flag == B_WRITE) ? U_WUD : U_RUD;

    if (copyio((caddr_t) cp, u.u_base, cnt, dirflag) == -1) {
          u.u-error = EFAULT;
          return;
    }
    u.u_base   += cnt;
    u.u_offset += cnt;
    u.u_count -= cnt;
}
```

1.2.2 Modes of Operation

When a process is executing instructions in the user program, it is said to be in *user mode*. When it is executing instructions in the XENIX kernel, it is said to be in *system mode*. When the kernel receives an interrupt from an external device, it switches to system mode if it was in user mode and passes control to the interrupt routine of the appropriate device driver. When the driver is done, it returns control of the process to the kernel, and the processing that was interrupted is resumed. The processing that took place as a result of 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 XENIX 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, XENIX 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 the CPU, and the kernel may decide to switch control, or *context* (described in the following section), to a different process.

1.2.3 Context Switching

Context switching occurs when the kernel decides to transfer 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.
- The 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 calls the kernel routine **sleep**().
- An interrupt is received that lets a blocked process proceed. This case will occur when the process has been sleeping at high priority, waiting for the interrupt handler to call **wakeup** 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

will occur.

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.

1.2.4 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' normal 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. 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 contained in the u-area.

The XENIX operating system (and therefore the task-time portions of the device drivers) uses a fixed-size, system-mode stack in the u-area. In XENIX, the size of this per-process stack is 1024 bytes. It is critical, then, that device-driver procedures not create local (stack) buffers of any significant size. If a device driver uses, or causes to be used, more than 1024 bytes of stack space, the kernel panics with a stack overflow. It is especially important that interrupt routines use little or no stack space. The following declaration will cause trouble, since as soon as the routine is called it requires at least 1024 bytes of stack space:

```
open()
{
        char buf [512];
        char buf2[512];
}
```

Further, interrupt-service 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.

1.2.5 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 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*.

On a multitasking system like XENIX, 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 *Guide*.

Each time a driver is invoked, it services only the specific system call that the user process requested. The active process' u-area is always mapped into the kernel's address space, 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 blocks, awaiting some event, the system scheduler schedules some other process, which may be in either user or system mode. This is commonly known as a "context switch." In other words, 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. These task-time portions of the driver can reference and perhaps modify the u-area, since the currently running process' u-area is always mapped into the kernel's address space.

1.2.6 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 memory.

The task-time portion of the device driver keeps the local (frame) 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. It may be important that the transfer take place directly into user memory, since it is necessary to lock the user program into physical memory to prevent swapping.

Typically, the task-time portion of the device driver issues a **sleep** call when it makes the initial I/O request. The interrupt-service routine 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** 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 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-memory locations.

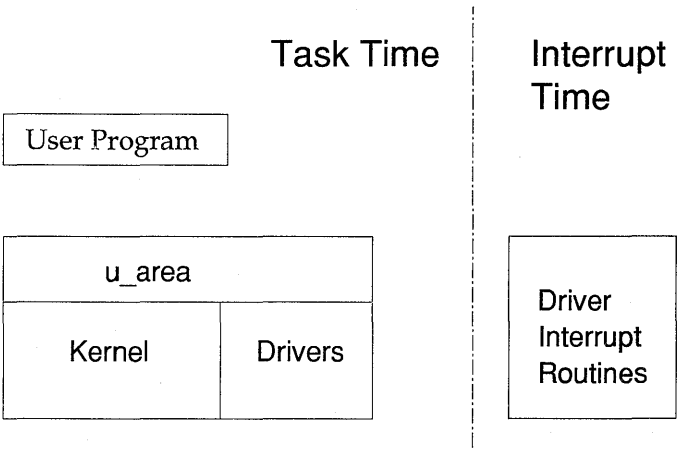
Access to static variables that can be modified at interrupt time is interlocked with the system priority level routines. These routines raise the interrupt priority of the CPU so that interrupts that might cause a value to change are locked out until the **splx** routine is called. This period must be kept as short as possible. For a more detailed description of these

routines, see the section titled "Interrupt Support Routines."

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 */usr/sys/h/iobuf*. The header for the buffer associated with the current transfer is defined in */usr/sys/h/buf.h*. Standard character I/O device drivers use the per-device tty structure (defined in */usr/sys/h/tty.h*) to pass information about the I/O request.

1.2.7 Interrupt Service Routine Rules

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 relationship between the scope of task-time and interrupt-time routines is illustrated in the following figure:



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 frame variables and buffers to devices and interrupt-service routines. Those locations are valid only when that individual user process is executing.

1.3 Parameter Passing to Device Drivers

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 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

To determine the values to use for *u.u_segflg*, see the */usr/sys/h/user.h* file. In addition to the parameters passed in the u-area, the kernel I/O routines pass the minor device numbers as parameters 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. For more information on using the buffer header information to set up a block data transfer, see the section on "Device Drivers for Block Devices."

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. For more information on using clists and the character I/O interface routines, see the section on "Device Drivers for Character Devices."

1.4 Naming Conventions

There are naming conventions for all driver routines called by the kernel, and for some driver variables. Each driver uses a unique two-to-four character prefix to identify its routines. For example, a hard-disk driver might use the prefix *hd*. In the following sections, the prefix used is *xx*.

Chapter 2

Block Device Drivers

2.1 Introduction to Block Devices

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

Unlike character devices, a block-I/O-transfer request is not a private transaction between a driver and a user process. The XENIX kernel provides a comprehensive buffer-management scheme that is used by block-device drivers.

The XENIX kernel maintains a pool of buffers, and keeps track of what data is in them, and whether the block 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.

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 will have 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.

2.1.1 Character Interface to Block Devices

Sometimes block-device drivers provide a character-I/O interface as well as one for block I/O. In this case, you can create a separate special-device file to access the device through the character interface. To construct a character-I/O interface to a block device, use the **mknod**(C) utility to create a character special device file that has the same major and minor number as the block special-device file for this device. The block-device driver must provide the **xxread** and **xxwrite** routines to implement

character I/O. For more information about creating this device file, see *mknod*(C) in the *XENIX Reference Manual.*

When a block device is accessed through a character interface, data transfer takes place directly between the device and the process' memory space. There is no intermediate buffering in the kernel-buffer pool or the clists. 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. This type of data transfer is referred to as physical, or raw, I/O. It 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. The **physio** routine, called by the **xxread** and **xxwrite** driver routines, locks the process in memory (core) for the duration of the data transfer.

2.1.2 Block Device Driver Routines

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

The following group of routines must be supplied by the device driver writer:

xxinit(), xxopen(), xxclose(), xxstrategy(), xxstart(), xxintr(), xxread(), xxwrite(), xxioctl()

The second group of routines is supplied by the kernel:

physio(), brelse(), deverr(), disksort(), getablk(), iodone(), iowait()

A block device appears to the kernel as a randomly addressable set of records of size BSIZE, where BSIZE is a manifest constant defined in the *sys/param.h* file. The XENIX 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 */usr/sys/h/buf.h* describes the fields in the buffer header. The fields most relevant to the device driver are:

Field	Contains
b_dev	The major and minor numbers of the device
b_bcount	The number of bytes to transfer
b_paddr	The physical address of the buffer
b_blkno	The block number on the device
b_error	Set if an error occurred during the transfer

The driver validates the transfer parameters in the buffer header, and then queues the buffer on a doubly linked list of pending requests. In each block-device driver, a header named *xxtab* (of type *struct iobuf*) points to this chain of requests. The */usr/sys/h/iobuf.h* file describes the fields in the request-queue header. The requests in the list are kept sorted using the **disksort** 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, and 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.

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 XENIX is first booted. If present, it is called indirectly through the *dinitsw* table defined in the kernel configuration file */usr/sys/conf/c.c*. It is also good practice for the **xxinit** routine to announce the presence of the device it is associated with.

Syntax: **xxopen(dev,flag,id)**
int, dev, 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 value of *dev* is an integer that specifies the minor device number.

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

The value of *id* is an integer that specifies whether the device is a block device (1) or a character device (0).

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

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

Parameters: The value of *dev* is an integer that specifies the minor device number.

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

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

Description: The kernel calls the **xxstrategy** routine to queue an I/O request. It must make sure the request is for a valid block, and then insert the request into the queue. Usually the driver calls **disksort()** 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.

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

Syntax: **xxstart ()**

Description: If the task-time portion of the driver detects that the device is idle, the **xxstart** routine may start it. It is often called by both task-time and interrupt-time parts of the driver. It checks whether the device is ready to accept another transfer request. If it is, the **xxstart** routine starts it up, usually by sending it a control word.

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, 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 will provide a character-device-driver interface 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 of arbitrary size, or read large portions of a disk directly. When a block device is referenced through the character-device interface, it is called *raw I/O* to emphasize 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)**
int dev;

Description: The **xxread** routine calls the **physio** with the appropriate arguments. This is the only action **xxread** performs.

Parameters: The value of *dev* is an integer that specifies the minor device number.

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

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

Parameters: The value of *dev* is an integer that specifies the minor device number.

Syntax:
xxioctl(dev, cmd, arg, mode)
int dev, cmd, mode;
faddr_t arg;

Description: The kernel calls the **xxioctl** routine when a user process makes an **ioctl** 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* is an integer that specifies the minor device number.

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

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

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

2.1.3 Kernel Provided Routines

The following routines are provided by the kernel for the device driver writer.

Syntax:
physio(strategy, bp, dev, flag)
int (*strat) ();
struct buf *bp;
int dev, flag;

Description: The **physio** routine provides the raw (direct) I/O interface for block-device drivers. It validates the request, builds a buffer header, locks the process in core, and calls the strategy routine to queue the request.

Note If the data transfer crosses a 64K segment boundary, **physio** may break the request into 3 pieces. On a 386, if the data request crosses a 4K page boundary, the request is broken into **BSIZE** pieces.

Parameters: The *strategy* argument specifies a pointer to the disk-strategy routine for the block device.

The *bp* argument specifies a pointer to the buffer header describing the request to be filled.

The value of *dev* is an integer that specifies the minor device number.

The *flag* argument specifies the I/O operation to be performed. The following table describes the flags this routine accepts:

Flag	Operation
BREAD	Reads from disk to user memory
BWRITE	Writes from user memory to disk
B_TAPE	Sets if driver is transferring variable size records (not multiples of BSIZE)

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. These values must not be odd. If B_TAPE is not set, all transfers must be multiples of BSIZE.

Syntax:
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.

Parameters: The *bp* argument is a pointer to the buffer header relating to the buffer to be released.

Return: The buffer addressed by *bp* is returned to the free buffer pool. No errors are possible.

Syntax:

```
deverr (dp, o1, o2, dev)
struct iobuf * dp;
int o1, o2;
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. The exact format of the output is shown in the following **printf** statement:

```
    register struct buf *bp;

  bp=dp->b_actf;
  printf("error on dev %s (%u/%u)",
  dev,
  major(bp->b_dev),
  minor(bp->b_dev));
  printf(", block=%D cmd=%x status=%x0,
  bp->b_blkno,
  o1, o2);
```

Parameters: The *dp* argument is a pointer to the head of the I/O request queue for the device.

The *o1* argument contains driver-specific information. It is normally used to provide the controller command that relates to the I/O operation that failed.

The *o2* argument contains driver-specific information. It is normally used to provide the controller status information at the time of failure.

The *dev* argument is a pointer to a string containing the device name.

Syntax:

```
disksort (disktab, bp)
struct iobuf * disktab;
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-strategy routine normally calls **disksort**. The *disktab* 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 in an attempt to reduce disk head movement.

Parameters: The *disktab* parameter is the address of a data structure **iobuf** declared within the driver to form the head of the I/O request queue.

The *bp* argument is a data structure **buf** * that points to the I/O request to be added to the queue.

Syntax:

```
struct buf *
getablk (flag)
int flag;
```

Description: The **getablk** 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**() or **iodone**().

Parameters: For XENIX-286, the values of *flag* are:

Value	Results
0	Returns any buffer
1	Returns only a system-addressable buffer
2	Returns a buffer that is guaranteed not to be a system-addressable buffer

The value of *flag* is ignored for XENIX-386.

Return: The routine returns a **struct buf** * that addresses the allocated buffer.

Warning for 286 only

There are very few directly addressable buffers in the XENIX kernel. Most are already allocated for other functions. If a directly addressable buffer is required, the value of SABUF may have to be increased in the *master* file.

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

Description: The **iodone** routine signals completion of an I/O operation involving the buffer addressed by *bp*. This routine is called when the driver wishes to signal either successful or erroneous completion of an I/O operation. It differs from the **brelse**() routine in that the higher levels of the kernel I/O system will complete the processing of the buffer before releasing it back to the buffer pool using **brelse**().

Parameters: The *bp* argument specifies a **struct buf** * that addresses the buffer.

Syntax: **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** routine.

Parameters: The *bp* argument specifies a **struct buf** * that addresses the buffer involved in the I/O operation.

Return: There is no result returned. The calling process will be allowed to proceed once the I/O operation has been completed.

Chapter 3

Character Device Drivers

3.1 Introduction to Character Devices

This section describes XENIX character device drivers. Character devices conform to the XENIX file model. Their data consists of a stream of bytes delimited only by the beginning and end of file. The XENIX system provides programs with direct access to devices through the special-device files. For more information on special-device files, see the section on "Special-Device Files."

Most character-device drivers in XENIX should be 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. For more information about this process, see the section on "Character-List and Character-Block Architecture."

3.1.1 Character Device-Driver Routines

xxinit(), xxopen(), xxclose(), xxstart(), xxhalt(), xxintr(), xxread(), xxwrite(), xxpoll(), xxproc(), xxioctl(), cpass(), passc()

Note that in the above list of routines, all the routines beginning with *xx* are user-supplied driver routines. However, both **cpass()** and **passc()** are standard routines supplied with XENIX.

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 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 nominative that refers to the device type. For example, a mouse driver would begin its routines with "mous"

type. For example, a mouse driver would begin its routines with "mous" rather than "xx." Since these routines are universally used, we substitute the characters "xx" when dealing with the generic routines:

Syntax: **xxinit ()**

Description: The **xxinit** routine initializes the device when XENIX is first booted. If present, it is called indirectly through the *dinitsw* table defined in */usr/sys/conf/c.c*, the kernel-configuration file. It is also good practice for the **xxinit** routine to announce the presence of its device.

Syntax: **xxopen (dev, flag, id)**
int, dev, 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 value of *dev* is an integer that specifies the minor device number.

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

The value of *id* is an integer that specifies whether the device is a character device (0) or a block device (1).

Syntax: **xxclose (dev, flag)**
int dev, 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* is an integer that specifies the minor device number.

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

Syntax: **xxstart ()**

Description: If the task-time portion of the driver detects that the device is idle, the **xxstart** routine can be called to start it. This routine checks whether the device is ready to accept another transfer request, and if so, starts it up, usually by sending it a control word. It is often called by both task-time and interrupt-time parts of the driver. The **xxstart** routine is 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. That is 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 signals 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.

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)**
int dev;

Description: The **xxread** routine is called when a program makes a read system call. The **xxread** subroutine transfers data to the user's address space. A subroutine, **passc**, can transfer one character at a time to the user. This subroutine returns a -1 when there are no more characters to be transferred.

Parameters: The value of *dev* is an integer that specifies the minor device number.

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

Description: The **xxwrite** routine is called when a program makes a write system call. This routine transfers data from the user's address space. A subroutine, **cpass**, can transfer one character at a time from the user. This subroutine returns a -1 when there are no more characters to be transferred.

Parameters: The value of *dev* is an integer that specifies the minor device number.

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

Description: The **xxpoll** routine, if present, is called by the system clock at **spl6()** 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*), defined in */usr/sys/h/param.h*, can be used to determine if the interrupted process was executing in user mode.

Warnings Do not call **xxpoll()** if your interrupt priority level is set at spl5 or higher. If you do, you will miss the interrupts at spl6 described above.

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

Description: The **xxproc** routine performs output-character expansion, outputs characters, and halts or restarts character output, according to the desired change in the output.

Parameters: The *tp* argument specifies the tty value of the device.

The *cmd* argument specifies the process to be performed. For a list of *cmd* arguments that are accepted by the **xxproc** routine, see the "Example Driver Code." chapter.

Syntax:
```
xxioctl (dev, cmd, arg, mode)
int dev, cmd, mode;
faddr_t arg;
```

Description: The kernel calls the **xxioctl** routine when a user process makes an **ioctl** 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* is an integer that specifies the minor device number.

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** system call for the device.

3.1.2 Kernel Provided Routines

The following routines are provided by the kernel to support character device drivers.

Syntax: **int cpass ()**

Description: The **cpass** routine returns the next character in a user output request. This function is provided by the kernel and does not need to be written by the device driver writer.

Return: The routine returns a character or the value -1, which indicates that there are no characters left in the output request.

Syntax:
int
passc (c)
int c;

Description: The **passc** routine passes characters to a user read request. This function is provided by the kernel and does not need to be written by the device driver writer.

Parameters: The character *c* is passed to the read request.

Return: The routine returns 0 normally and -1 when the user read request has been satisfied.

Serial Driver Support Routines

This section describes the routines provided by the kernel to initialize the tty structures, start the tty output, and empty the tty queue. The tty structure is defined in */usr/include/sys/tty.h*. These routines are used almost exclusively by serial drivers.

emdupmap(), emunmap(), ttinit(), ttiocom(), ttstrt(), ttyflush()

Syntax: **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: This routine has no return value.

Syntax: **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: This routine has no return value.

Syntax: **ttinit(tp)**
struct tty *tp;

Description: This routine initializes the *tty* structure to specific default values. To set up the default settings for a *tty* device, call this routine immediately after opening the *tty* device. *ttinit* initializes the *t_line*, *t_iflag*, *t_oflag* , *t_cflag*, *t_lflag*, and *t_cc* fields of the *tty* structure.

Parameters: *tp* is a *struct tty** that points to the *tty* data structure associated with the device being used.

Syntax: **ttiocom(tp, cmd, addr, flag)**
struct tty *tp;
int cmd;

Description: This routine is called for all common *tty ioctl* calls. It is called by the *xxioctl( )* routine after a device specific *ioctl* has been performed. **Parameters:** *tp* is a *struct tty** that points to the *tty* data structure associated with the device being used.

cmd is an integer specifying an *ioctl* command.

addr specifies the address of the user space where the parameters reside.

flag specifies whether the command is a read or write operation.

Syntax: **ttrstrt(tp)**
struct tty *tp;

Description: This routine restarts *tty* output after a *timeout( )* call. It is passed as an argument by the device driver to *timeout( )* calls.

Parameters: *tp* is a *struct tty** that points to the *tty* data structure associated with the device being used.

Syntax: **ttyflush(tp, cmd)**
struct tty *tp;

Description: This routine flushes the *tty* queue.

Parameters: *tp* is a *struct tty** that points to the *tty* data structure associated with the device being used.

cmd specifies whether to flush the input (FREAD) queue or the output (FWRITE) queue. (FREAD) and (FWRITE) are defined in */usr/sys/h/file.h*

Character List Routines

The kernel contains a group of small buffers called character lists, or *clists*. A *clist* structure is the head of a linked list queue of characters. The elements in the linked list are called *cblocks* and each *cblock* can hold a small number of characters. These are used for buffering low-speed character devices.

Drivers that do not use the *tty* structure must declare a queue header of type *clist*, or two queue headers if both input and output are to be buffered. The *tty* structure already contains declarations for the needed queue headers. There are eight routines that the driver can use to manipulate *clist* buffers, as described below. All these routines can be used during interrupt-time processing.

Syntax: **getc(cp)**
struct clist *cp;

Description: This routine moves one character from the *clist* buffer for each call.

Parameters: *cp* specifies the *clist* buffer from which characters are moved.

Return This routine returns the next character in the buffer or -1 if the buffer is empty.

Syntax: **putc (c, cp)**
int c;
struct clist *cp;

Description: The **putc** routine moves one character to the clist buffer for each call.

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.

Return: This routine returns 0 if it places the specified character in the buffer, or returns -1 if there is no free space.

Syntax:
```
struct cblock*
getcb (cp)
struct clist *cp;
```

Description: The **getcb** routine moves one cblock from the clist buffer for each call.

Parameters: The pointer *cp* specifies the clist buffer the cblocks are moved from.

Return: This routine returns a pointer to the first cblock onthe clist or NULL if the clist is empty.

Syntax:
```
getcbc (bp)
struct cblock *bp;
```

Description: The **getcbc** routine returns the first character from the cblock pointed to by *bp*.

Parameters: The pointer *bp* specifies the cblock the character is taken from.

Return: This routine returns the first character from the cblock that *bp* points to.

Syntax:
```
getcbp (p, cp, n)
struct clist *p;
char *cp;
int n;
```

Description: The **getcbp** routine copies characters from the specified clist, *p*, to the buffer addressed by the *cp*

argument.

Parameters: The pointer *p* specifies the clist buffer the characters are copied from.

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: This routine returns the number of characters actually copied (which is less than or equal to *n*).

Syntax:
struct cblock *
getcf ()

Description: The **getcf** routine takes a cblock from the freelist and returns a pointer to it.

Return: Returns a pointer to a cblock if available. Otherwise, the routine returns NULL.

Syntax:
putcb (cbp, cp)
struct cblock *cbp;
struct clist *cp;

Description: The **putcb** routine moves one cblock to the clist buffer for each call.

Parameters: The pointer *cbp* specifies the cblock to be moved.

The pointer *cp* specifies the clist buffer to where the cblock is moved.

Return: This routine returns 0 after placing the specified cblock in the buffer.

Syntax: **putcbp (p, cp, n)**
struct clist *p;
char *cp;
int n;

Description: The **putcbp** routine copies characters from a buffer to the clist given as an argument.

Parameters: The pointer *p* specifies the clist buffer the characters are copied to.

The argument *cp* is a *char* * that addresses the buffer the characters are copied from.

The value of *n* is the number of characters to be copied to the *clist.*

Return: This routine returns the number of chracters transferred. This is equal to *n* if there was sufficient room in the clist or less than *n* if the clist was filled before the transfer was complete.

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

Description: The **putcf** routine puts the specified cblock onto the freelist.

Parameters: The argument *cbp* specifies a pointer to a cblock.

Note

- All the cblocks not currently being used are kept on a list of free memory blocks. Since there are a limited number of cblocks in the system, each driver must be judicious in determining how many cblocks are used for buffering input and output.

- For output buffering, the driver usually follows a *high-* and *low-water mark* convention. The driver accepts and queues requests from the user process until the buffer has reached its high-water mark. At that point, the requesting processes are suspended by means of the **sleep** routine. When the buffer has drained below the low-water mark, the suspended processes are awakened, and can fill the buffer again.

- For input buffering, the driver usually buffers the data up to some limit. When this limit is reached, data is discarded to make room for the more recent data.

Line-Discipline Routines

If you use a serial device as an interactive terminal, it must support various functions such as erasing characters and lines, echoing, and buffering input. The code needed to perform each of these functions has been separated into a set of routines that roughly corresponds to the character-device function. Each of these routines is called a *line discipline*. One standard line discipline is provided by default. Each of the routines is called through the *linesw* table initialized in */usr/sys/conf/c.c*; each entry in this table represents one line discipline, and has entries for eight functions.

The **l_open** routine is called on the first open of a device. The **l_close** routine is called on the last close of the device. The **l_read** and **l_write** routines are called by the driver's read and write routines, to pass characters to and from the calling process. The **l_input** routine buffers incoming characters at *interrupt time*. The **l_ioctl** routine calls specific routines related to line discipline. The **l_output** routine gets the next block of characters for output at interrupt time. The **l_mdmint** routine is not currently used.

3.1.3 Interrupt Routines for Character-Device Drivers

The device-interrupt routine is entered whenever one of its devices raises an interrupt. Note that in general one driver may control several devices, but that all interrupts are vectored through a single function-entry point, usually called **xxintr**, where *xx* is a mnemonic that refers to the device type (see the Section on "Naming Conventions"). 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** routine will alert 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 clists each time a transfer completes.

3.1.4 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.

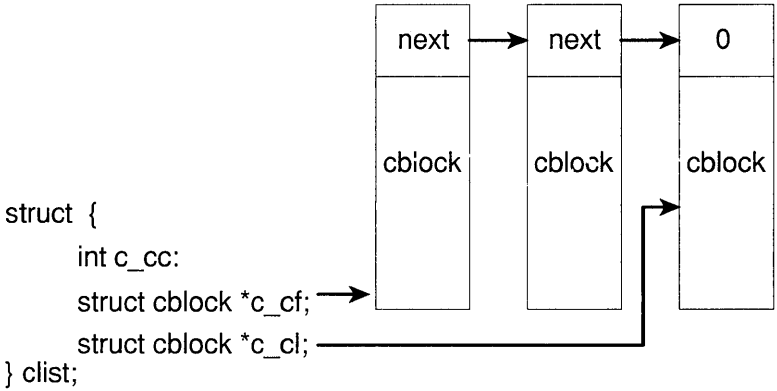
The XENIX kernel has a pool of character blocks called *cblocks*. Each cblock contains a link to the next cblock and an array of characters. A clist is a linked-list queue of cblocks.

The **getc** and **putc** kernel routines put characters into and remove characters from a clist. Drivers using clists 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
- a pointer to the first cblock in the list
- a pointer to the last cblock

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



There is a protocol defined for use of 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 *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.

3.1.5 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 queue*, the *canonical queue*, and the *output queue*.

When a process writes data to a terminal device, the task-time part 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 XENIX 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, XENIX 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

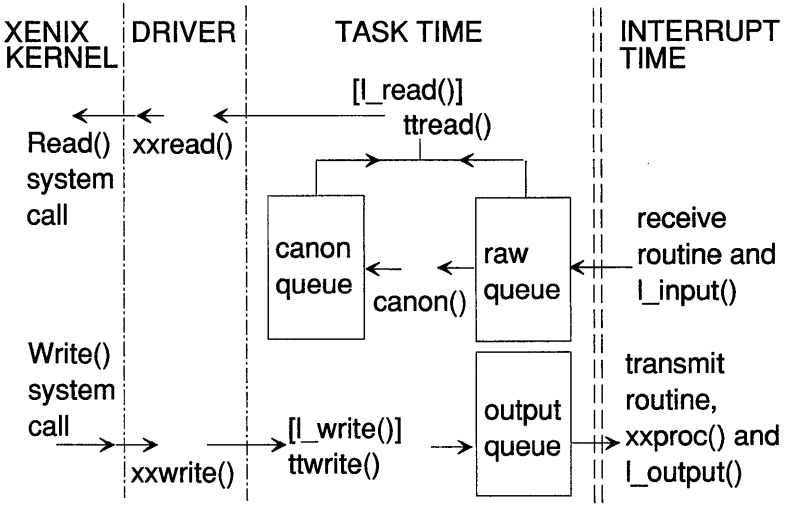
processing. At task time, the driver decides how much processing to do. The user process has the option of requesting *raw* input, in which it receives data directly from the raw queue. *Cooked* (the opposite of raw) input refers to the input after processing for ERASE, line kill, DELETE, and other special treatment. In this case, a task-time routine, **canon**, 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** system call.

In XENIX System V, the direct clist processing for tty device drivers is normally handled by the specific line discipline. 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. For more information about code, see the "Example Driver Code" chapter.

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 uppercase 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.

3.1.6 Other Character Devices

The following character devices are commonly found on XENIX systems:

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

The system console has a large section of dedicated kernel code to handle its special needs. See the following section on "Writing a Video Adapter Driver" for more information on the system console and multiscreens.

Terminals receive a lot of special attention in the XENIX system. Line printers and magnetic-tape drivers 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 Drivers

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.

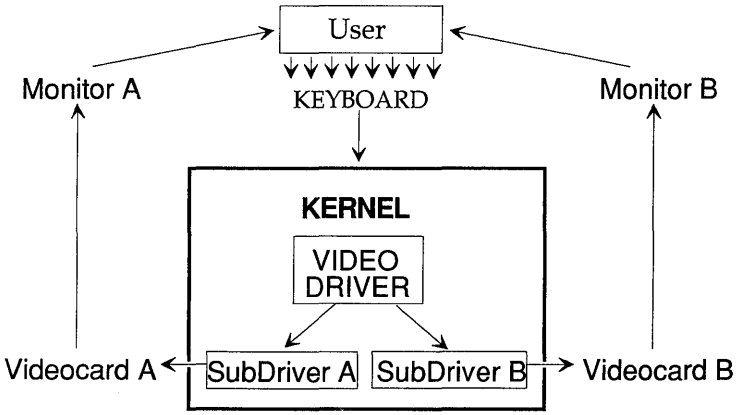
Usually tape drivers provide two interfaces: a *block* and a *character* interface. The character interface is used for raw, or physical, I/O directly between the device and the user process' address space. The block interface makes use of the XENIX kernel-buffer pool and buffer-manipulation routines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

Chapter 4

Video Device Drivers

4.1 Writing a Video Adapter Driver

The XENIX video driver is designed to make use of sub-drivers written for individual video hardware. The top level of the video driver is provided with your kernel, along with a group of standard monochrome, CGA, EGA, and VGA sub-drivers. If you wish to use a different or non-standard video card, you must provide a sub-driver for that card. The following example shows the relationship between the user, the kernel, the video driver, the sub-driver, and the hardware:



There is a predefined set of entry points for video subdrivers. These entry points are eleven routines that must be provided by the device driver writer. These eleven routines include three initialization routines, called xxcmos, xxinit, and xxinitscreen. Secondly, there are four data handling routines, called xxscroll, xxclear, xxcopy, and xxpchar. Finally, there are four special routines that you must provide. These are xxscurs, xxioctl, xxsgr, and xxadapctl. As with all device drivers, you replace the prefix "xx" with the unique two to four letter identifier for your particular driver. As previously stated, there are supplied sub-drivers for standard monochrome, CGA, EGA and VGA adapters. The data structures and entry points for these and sub-drivers are described in the file */usr/sys/io/cnconf.c*.

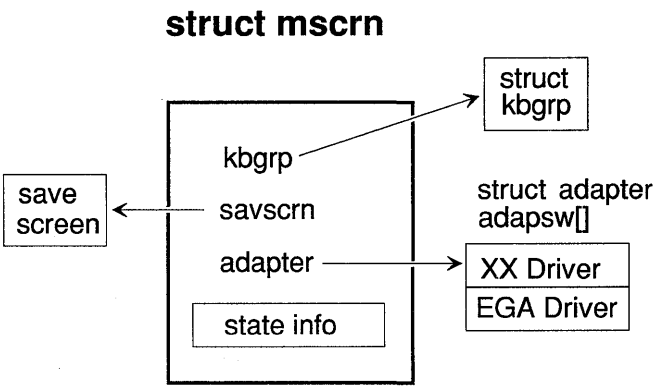
Video Driver Data Structures

The Video driver provides three data structures for use with sub-drivers. These are:

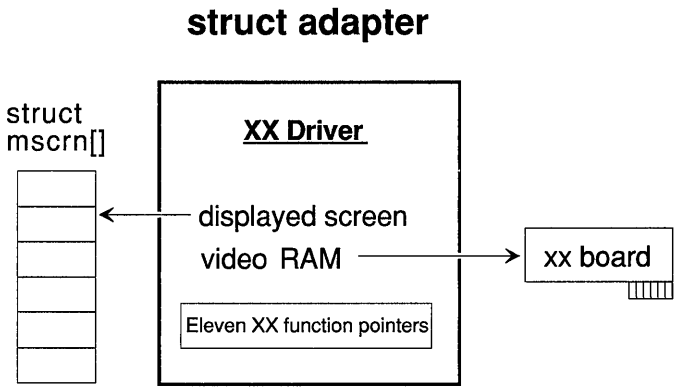
```
struct mscrn
struct adapter
struct kbgrp
```

struct mscrn is a structure that is replicated for each individual multiscreen. Each structure contains state information for the multiscreen (current color, mode, etc.) and a series of pointers. The complete definition of this structure can be found in the file */usr/include/sys/vid.h.*

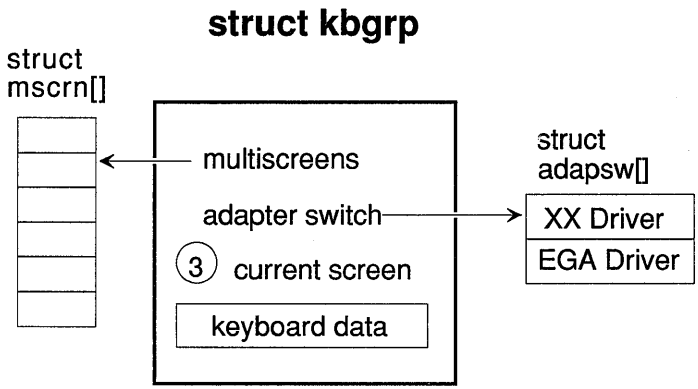
There is a pointer to **struct kbgrp**, the keyboard group that the multiscreen belongs to. Note that all multiscreens on the main system monitor will have the same keyboard group. There is also a pointer called **mv_savscrn** that points to the memory location where the screen image is saved when the multiscreen is not active. And finally, there is a pointer to **struct adapter**; the structure containing information including the entry points into the adapter routines that manage the monitor on which the multiscreen appears. Here is a graphic representation of the relationship between **struct mscrn** and the other parts of the driver:



struct adapter is a structure that is replicated for each sub-driver and contains pointers to each of the eleven routines that a sub-driver should have, and other pointers to the **struct mscrn** of the current multiscreen and to the video RAM of the card that the sub-driver drives. The following is a graphic representation of the relationship between **struct adapter** and the other hardware and software that comprise the video system:



struct kbgrp defines a keyboard group. This structure is replicated for every keyboard attached directly to your computer. (i.e. keyboards on terminals attached via serial lines are not included.) **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. The following picture is a graphic representation of the relationship between **struct kbgrp** and the other software fo the video driver:



4.1.1 Video Sub-Driver Routines

Here is a close look at the eleven routines you must provide to create a video sub-driver.

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

Purpose: This routine should read the hardware characteristics and specify whether the video card in question is to be the primary card. 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 read the switches on the card.

Parameters: The pointers ***prip** and ***secp** point to the primary and secondary video devices to be used. Your driver can assign itself as the primary device by assigning its value from the adapter structure **r_type** into ***prip**. Likewise, you can also assign your driver to be secondary by assinging 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 once per multiscreen. Any initialization that needs to take place on a per-screen basis should happen here.

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;
int *bp;
int cnt;

Purpose: This routine writes data beginning at the current cursor position.

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 moves the cursor from one multiscreen to another.

Parameters: ***msp** points to the new multiscreen receiving the cursor.

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

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. **gr** is the ANSI code for the desired graphics rendition. Standard values of **gr** that are passed to your xxsgr routine are:


```
SGR_NORMAL      0   /* return attributes to normal */
SGR_BOLD        1   /* called INTENSE in video'ese */
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 */

    /* Fonts 0, 1, and 2 are reserved */

SGR_FONT3       13  /* alternate font #3 */
SGR_FONT4       14  /* alternate font #4 */
SGR_FONT5       15  /* alternate font #5 */
SGR_FONT6       16  /* alternate font #6 */
SGR_FONT7       17  /* alternate font #7 */
SGR_FONT8       18  /* alternate font #8 */
SGR_FONT9       19  /* alternate font #9 */
SGR_FOREBLACK   30  /* ANSI foreground colors */
SGR_FORERED     31  /* ANSI foreground colors */
```

```
SGR_FOREGREEN       32   /* ANSI foreground colors */
SGR_FOREYELOW       33   /* ANSI foreground colors */
SGR_FOREBLUE        34   /* ANSI foreground colors */
SGR_FOREMAGENTA     35   /* ANSI foreground colors */
SGR_FORECYAN        36   /* ANSI foreground colors */
SGR_FOREWHITE       37   /* ANSI foreground colors */
SGR_BACKBLACK       40   /* ANSI background colors */
SGR_BACKRED         41   /* ANSI background colors */
SGR_BACKGREEN       42   /* ANSI background colors */
SGR_BACKYELLOW      43   /* ANSI background colors */
SGR_BACKBLUE        44   /* ANSI background colors */
SGR_BACKMAGENTA     45   /* ANSI background colors */
SGR_BACKCYAN        46   /* ANSI background colors */
SGR_BACKWHITE       47   /* ANSI background colors */
```

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

Purpose: This routine provides support for any ioctls you may want to support. In practice, you can use this function to support standard ioctls to 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 MAP_CONS. These stand for CONSOLE GET and MAP CONSOLE. **arg** is any arguments to the ioctl command. **mode** is the new mode (such as graphics vs. 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
AC_DEFRF       13    /* define reverse foreground */
AC_DEFRB       14    /* define reverse background */
AC_ONR              15
AC_DEFGF       16    /* define graphic foreground */
AC_DEFGB       17    /* define graphic background */
AC_ONG              18
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 IO privl */
AC_SOFTRESET  107    /* reset text mode (keep colors)*/
AC_SENDSCRN   108    /* write screen chars to stdin */
AC_VTKDPARAM  200    /* get VPIX display parameters */
AC_TXTRECVR   201    /* recover text mode from DOS */
AC_TXTRELSE   202    /* release text mode to DOS */
```

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 5

Compiling and Linking Drivers

5.1 Compiling, Configuring, and Linking Drivers

To make your driver source code part of the XENIX kernel, you first compile your driver in the same way that the rest of the kernel is compiled.

Next, make the various routine names and driver attributes of your driver accessible to the XENIX kernel using the **configure** utility. This program creates several multi-dimensional tables of routine names and driver attributes.

Then, link the kernel by adding the new module to the **ld**(C) command line in the *makefile* provided, and running **make**(CP).

5.1.1 Compiling Device Drivers

Use the XENIX 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:

-K	Disable stack probes.
-DM_KERNEL	Required for conditional code in standard header files

It should also contain ONE of the following:

-M3	For 80386 processors.
-M2em	For 80286 processors. Enables 80286 instructions, **near**, and **far** keywords. Compiles middle model, to conform with the kernel program model.
-M0em	For 8086 processors. Uses only 8086 instructions. Enables **near** and **far** keywords. Compiles middle model, to conform with the kernel program model.

For device driver subroutines written in assembly 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 will produce a corresponding ".o" module. For example, *scsi.c* becomes the object module *scsi.o*.

5.1.2 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.

The **configure** utility generates and assembles the *c.asm* and *space.inc* source modules that contain these indirect function references. The generated *space.inc* file is one of the header files that is inserted into *space.asm* before that module is assembled into *space.o*. The *c.asm* file is assembled into the object module *c.o*. Both object modules are then linked into the kernel.

Some older "preconfigured" drivers did not require **configure** to be run, as all function references were already in place. For the rest, composing the driver's configuration command is discussed in **configure**(ADM), *Using the Link Kit*, and briefly, below. Though it may seem easier to edit the C and assembly language configuration files directly, **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.

Preconfigured Drivers

Earlier releases made provisions for several common types of drivers by providing null routines that were linked in if the corresponding drivers were not present. Drivers for which preconfiguration was provided will still link in as before. Simply add the name of the driver's ".o " module to the **ld** command line, as before. The earlier scheme of patching the interrupt vector within *vecintsw* at driver initialization time should still be used.

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

For the 80386 or 80286 mapped kernel, your peripheral device can interrupt on one of the request lines of either a master interrupt controller or single slave interrupt controller, which is connected to master request line 2. Your XENIX 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.

2. If it is on a slave controller (only one is currently supported for the 80286 and 80386, on master request line 2), take the request line used on the slave controller and add octal 30. The result can then be used for your driver in */usr/sys/conf/master*.

For example, if your device uses request line 3 on the slave controller, you would specify 33 in the *master* file.

Vector Manipulation for Preconfigured Device Drivers

In preconfigured drivers, entry points have been provided for all necessary routines (open, close, etc.) except for the interrupt handler. This must be patched in at system start-up time as follows: an extra entry point has been provided for each of the drivers expected to require an interrupt vector. This entry point's suffix is "-init". This function must replace the appropriate vector in the *vecintsw*[ ] table with a pointer to the interrupt handler function for the particular device driver. This structure is declared in the file */usr/sys/h/conf.h*, and an example of its usage is in *c.c*.

In addition to patching the *vecintsw*[ ] table, the driver *init* routine should patch the *vecintlev*[ ] table. This table specifies the priority or *spl* level of the driver. Most drivers are priority level *spl5*.

An example driver fragment for a hypothetical *spl5* driver "xx" using bus vector 12 is shown below:

```
#include "../h/param.h"
#include "../h/conf.h"
    .
    .
    .
xxintr()
{
    .
    .
    .
}

#define   NUM   (030 + (12-8))

int (*xxoldintr())();

xxinit()
{
   xxoldintr = vecintsw[NUM];
   vecintsw[NUM]  = xxintr;
   vecintlev[NUM] = 5;
   .
   .
   .
   /* perhaps some other one-time */
   /* driver-local initialization */
}
```

It should be noted that this may not be applicable for all device drivers under all circumstances.

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 between 2 and 4 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 *XENIX 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. Maintain a backup copy of the *master* and *xenixconf* files while learning to use **configure**: if you make a mistake you can restore the old files.

Also note that **configure** requires that block drivers have a *tab* structure, and indeed, the vast majority of block drivers do. If you are writing a non-interrupting block driver, simply declare a *struct iobuf xxtab* within your driver.

5.1.3 Linking The Kernel

Your Link Kit contains a *makefile* for linking the kernel. The reference to the new driver should be placed in this file on the **ld** command line prior to any of the object library references (the pairs of options of the form **-l** *lib_xxx*), and following all other object file references. That is, your object file must follow *KMseg.o*, *oemsup.o*, *c.o*, and any other files already on the command line.

For binary distribution, also edit the file */usr/sys/conf/link_xenix* to link the new driver in with the kernel. Here too, place the reference to the new driver on the **ld** command line prior to any of the object library references and following all other object file references.

For preconfigured drivers, the **ld** command will find the actual driver first and thus stop looking for it in the libraries, which contain the null routines that are normally linked in.

To link your driver, enter:

```
make
```

link_xenix is what the end user uses to link your driver into the kernel. The end user may not have purchased the XENIX development system and therefore may not have **make**(CP). However, **ld**(M) comes with every system, and no special utility is required to run shell scripts.

Once you have a new XENIX kernel, back up the old one, by typing this or a similar command:

cp ./xenix /xenix.new

The new XENIX kernel must be in the / directory. Note that in some versions, a kernel must be one of the first 64 entries in the / directory for **boot** to find it.

5.2 Driver Debugging

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

5.2.1 Booting the New Kernel

Halt your system by entering:

/etc/haltsys

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

```
Boot
:
```

If you press RETURN, or simply do nothing, the default operating system image is loaded and started. In order for the bootstrap program to locate and load any newly installed device drivers, it must be told to read the */xenix.new* file, which contains the kernel that includes the device driver. To boot the new kernel, enter, at the boot prompt:

xenix.new

The system will boot up with the "new" kernel.

5.2.2 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, you might have a problem with either the device node itself, or with your driver configuration, as specified in the Link Kit configuration file *c.c*.

   Check your major device number correspondence. Make sure that the major device number and type of the device node you are trying to access corresponds with the appropriate line in the *_bdevsw* or *_cdevsw* array in *c.asm*. For example, you might have written a new printer driver, whose major device number is 6, whose name is "pa", and whose type is "character".

   First, check the device node, to make sure that /dev/pa is a character special device, whose **ls -l** listing is along the following lines:

   ```
   crw-rw-rw- 1 root      6,  0 Apr 29 19:56 /dev/pa
   ```

   Then, check *c.asm*, to make sure that there is a set of functions in the *_cdevsw* table that have the "pa" prefix. The file will be similar to this:

   ```
   DW $CFG_C6
   DD _paopen
   DD _paclose
   DD _nulldev
   DD _pawrite
   DD _paioctl
   DW 00H
   DW 00H
   ```

   The parallel driver has only the routines paopen, paclose, pawrite and paioctl as part of the *_cdevsw* table. Other drivers might have **read**, **_tty**, or in the future, **stream** entries.

   Also check the constants *_cdevcnt* and *_cdevmax* (*_bdevcnt* and *_bdevmax* for a block driver) to make sure they are at least 1 greater than the major number of your driver.

   If this correspondence does not hold, revert back to the older **master** and **xenixconf** files and rerun **configure**.

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

   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*( ) 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 you can write to one of your device's registers using *outb*( ), then read back the value you've written using *inb*( ). 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*( ) , 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*( ) statements for debugging.

   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. Coupled with #ifdef DEBUGs 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:");
    printf("dev=%x, flag=%x bc=%x",dev,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( )* 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( )* 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.

   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!

   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.

7. Use *spl{5,7}( )* as a debugging aid.

   Sometimes, a driver can be difficult to debug because higher priority interrupts get in the way. A call to *spl7( )* will shield you 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.

5.2.3 Vector Collision Considerations

When designing a device driver to work with XENIX, care should be taken in the selection of the hardware interrupt vector. This is because of the possibility of conflict between device drivers over interrupt vector usage.

8086-based XENIX systems use only one 8259 programmable interrupt controller. Of the 8 vectors available, only vector 2 (bus lead IRQ2) is not currently used. It is appropriate for devices whose drivers are written using *spl5*( ).

80286 and 80386-based XENIX 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, cohabitation is possible for those devices that interrupt only following a request from the CPU. Disk drivers, tape drivers, and other such devices can "time out", using the *timeout( )* function, 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.

5.2.4 Note on ps

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

See **ps**(C) in the XENIX *Reference* for more information.

5.3 Notes On Preparing a Driver for Binary Distribution

5.3.1 Naming Guidelines

The 2-4 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

TechnoBabble hard disk might begin its routines with the prefix "tbhd".

5.3.2 Style Issues for User Prompting

Most currently configured XENIX devices print out a short message in their initialization routines 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, etc.) are provided at boot time; you need only supply a line with information specific to your driver.

5.3.3 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. In System V, a driver's major device number is no longer passed to it as a parameter.

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>

extern struct bdevsw bdevsw[ ];

if ( bdevsw[major(rootdev)].d_open == xxopen ) {
  printf("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. An unused vector has a *vecintsw*[ ] entry of *novec*.

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.

5.3.4 Preparing Drivers to Use custom

The best thing you can do for the end user is to supply a driver installation shell script for use with **custom**(C). 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 magic 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 magic file has a name of the following form:

```
./tmp/_lbl/prd=sidd/typ=286AT/rel=1.0.0/vol=01
```

where *sidd* is the driver's prefix (in this case, it stands for Sample Installable Device Driver), and 286AT 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) The Santa Cruz Operation, 1985.
#     This Module contains Proprietary Information of
#     The Santa Cruz Operation, Microsoft Corporation
#     and AT&T, and should be treated as Confidential.
#
#prd=sidd
#typ=286AT
#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   ./usr/sys/conf/sidd.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 magic 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 driver 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)**.

The driver installation shell script has 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**
- Edit the name of the new driver into the shell script *link_xenix*
- Run *link_xenix* to link the kernel
- Make the device nodes in */dev*.
- Run the shell script *hdinstall*, which backs up the old *xenix*, and puts your new *xenix* in its place.

Here is a sample installation shell script for the aforementioned Sample Installable Device Driver. It must be extracted into */tmp*, and have a name that starts with "init."

```
#
#       Copyright (C) The Santa Cruz Operation, 1985, 1986.
#       This Module contains Proprietary Information of
#       The Santa Cruz Operation, Microsoft Corporation
#
#       Driver initialization script
#
PATH=/bin:/usr/bin:/etc

cd /

# Get the permlist for the set containing the link kit package.
# Link Kit Release 2.0 is found in the "base" set; link kit release
# 2.1 and 2.2 is found in the "inst" set.

if [ -f /etc/base.perms ]; then
        PERM=/etc/base.perms
elif [ -f /etc/inst.perms ]; then
        PERM=/etc/inst.perms
else
        echo "Cannot locate /etc/base.perms or /etc/inst.perms" >&2
        exit 1
fi

# test to see if link kit is installed
until fixperm -i -d LINK $PERM
do      case $? in
        4)      echo "The Link Kit is not installed." >&2   ;;
        5)      echo "The Link Kit is only partially installed." >&2;;
        *)      echo "Error testing for Link Kit. Exiting."; exit 1;;
        esac
        # Not fully installed. Do so here
        while echo "Do you wish to install it now? (y/n) \c"
```

```
        do      read ANSWER
                case $ANSWER in
                Y| y)  custom -o -i LINK
                       break
                       ;;
                N| n)  echo "Drivers cannot be installed without the Link Kit.
                       exit 1
                       ;;
                *)     echo "Please answer 'y' or 'n'. \c"
                       ;;
                esac
        done
done
```

```
echo "adding device entry points"

cd /usr/sys/conf

# if the 'sidd' driver is present in the "master"
# file, "configure -j sidd" prints its major device
# number and returns 0.  if the driver is not
# present in "master", "configure -j sidd" prints
# an error and returns 1
#
# configure -j NEXTMAJOR returns the smallest
# available major device number.
#
# configure -m $major ...  adds the given device
# entry points to XENIX.
#
if major=`configure -j sidd`
then
      echo "Device entry points already configured"
else
major=`configure -j NEXTMAJOR`
configure -m $major -b -a siddopen siddclose siddstrategy siddtab || {
      echo "Cannot add device entry points to XENIX"
            exit 1
      }
fi

echo "adding sample driver to link line"

grep -s sidd.o link_xenix >/dev/null || {
      # add sidd.o to link line
      cp link_xenix link_xenix.00 || {
            echo "Cannot copy link_xenix" >&2
            exit 1
      }
trap "mv link_xenix.00 link_xenix; exit 1" 1 2 3 15
sed "s!c.o!& sidd.o!" link_xenix.00 > link_xenix || {
            echo "Cannot edit link_xenix" >&2
            mv link_xenix.00 link_xenix
            exit 1
      }
      trap 1 2 3 15
      chmod 700 link_xenix
}

echo "\nRe-linking the kernel ... \c"
if link_xenix; then
      hdinstall
echo "\nInstallable Device Driver installation complete.\n"

else

echo "\nLink failed, you will have to re-link the kernel\n"
```

```
fi

# make sample device nodes
/etc/mknod /dev/sidd00 b $major 0
/etc/mknod /dev/sidd01 b $major 1

exit 0
```

To summarize, the custom-installable driver installation floppy must contain a *permlist*, a magic product identification file, the object module (extract into /usr/sys/conf), and the driver initialization script. Like all **custom**-installable floppies, a driver installation floppy is otherwise a normal **tar** volume.

```
-rw-r--r--1 root    0 Dec  9 08:46 ./tmp/_lbl/prd=sidd/
typ=286AT/rel= 1.0.0/vol=01
-rw-r--r--1 root  669 Dec 10 20:15 ./tmp/perms/sidd
-rw-r--r--1 root 6157 Dec 10 18:59 ./usr/sys/confsidd.o
-rwxrwxr-x1 root 2097 Dec 10 20:14 ./tmp/init.sidd
```

Chapter 6

Memory Management Routines

6.1 Memory Management Routines

The first four routines in this section work with both 80286 and 80386 systems. These are **db_alloc**, **db_free**, **db_read**, and **db_write**. The rest of this section describes the following 286-specific and 386-specific XENIX memory management routines:

For 80286 only: dscralloc(), dscrfree(), dscraddr(), ldtalloc(), ldtfree(), mmudescr() mmuget(), mmufree()

For 80386 only: IS386(), cvttoaddr(), cvttoint(), mapphys(), unmapphys(), mapptov(), memget(), sptalloc(), sptfree()

286 and 386 Memory Management Routines

Syntax: **db_alloc(tdbptr, reqs)**
struct devbuf **tdbptr;
short reqs;

Description: The **db_alloc()** routine allocates memory that is physically contiguous. Contiguous memory is necessary for performing DMA transfers. Memory for all other uses should be allocated using standard memory allocation routines for your machine.

Parameters: **db_alloc()** can be used to allocate up to 30 blocks of contiguous memory at one time. However, each block must be the same size. For each block, a devbuf structure must be defined. The parameters to **db_alloc()** are a pointer to an array of the devbuf structures (one structure per requested block), and a short integer indicating the number of requested blocks.

The calling code must initialize two of the fields in the *devbuf* structure: "size" and "ldbs." "ldbs" must be set to the value 9, and "size" should be set to the size of the requested block, in 512-byte units.

The following example code fragment allocates 2 buffers of 120K each:

```
struct devbuf tb[2];

tb[0].size = tb[1].size = 240;
```

```
/* 240 * 512 bytes = 120K bytes */
tb[0].ldbs = tb[1].ldbs = 9;

if( db_alloc( &tb, 2 ) == 0 ){
    printf( "Error: db_alloc() failed" );
    return;
}
```

Warning: This routine must not be used during the driver's initialization function. The **memget()** function can be called to obtain contiguous memory during driver initialization for the 386. **mmuget()** is used to obtain contiguous memory at initialization time for the 286. Reading from and writing to memory areas allocated using **db_alloc()** must be performed using the **db_read()** and **db_write()** functions only.

Syntax: **db_free(tdbptr, reqs)**
struct devbuf **tdbptr;
short reqs;

Description: This routine releases memory areas allocated via the **db_alloc()** function. The arguments are the same as those to **db_alloc()**. **db_free()** does not return any meaningful value.

Example: For example to free the memory allocated in the **db_alloc()**, you might use the following code:

```
struct devbuf tb[2];

db_free( &tb, 2 );
```

Syntax: **db_write(dbptr, phys_addr, count)**
struct devbuf *dbptr;
paddr_t phys_addr;
unsigned count;

Description: This routine is used to transfer data from the physical address pointed to by phys_addr to the memory pointed to by dbptr. The amount transferred is

"count" in bytes. This would most likely be used in the raw interface to a block device driver, so you would use the bp->p_paddr in the buf structure, with the count being bp->b_count.

Example: For example, to transfer from a buffer to a dbptr:

```
struct buf *bp;
struct devbuf dp;

db_write( &dp, paddr( bp ), bp->b_count );
```

Syntax:

```
db_read( dbptr, phys_addr, count )
struct devbuf *dbptr;
paddr_t phys_addr;
unsigned count;
```

Description: This routine is used to transfer data from the memory pointed to by dbptr to the physical address pointed to by phys_addr. The amount transferred is "count" in bytes. This could also be used in the raw interface of a block device driver, using bp->p_paddr in the buf structure as the phys_addr parameter, and bp->b_count as the count.

Example: For example, to transfer data from dbptr to a buffer:

```
struct buf *bp;
struct devbuf dp;

db_read( &dp, paddr( bp ), bp->b_count );
```


286-Specific Memory Management Routines

The memory management routines in this section are specific to XENIX-286. These routines access memory that is not within kernel data space (ie., to access memory-mapped devices). A descriptor from the Global Descriptor Table (GDT) can be initialized to map the memory area, and then used to access the memory.

Syntax:

```
unsigned short
dscralloc()
```

Description: The **dscralloc** routine allocates a descriptor from the pool of GDT descriptors available for drivers. It returns the 16-bit selector number of the allocated

descriptor. Note that this routine can be used only with XENIX-286.

Return: This routine returns 0 if no more descriptors are available, and prints the following message on the system console:

```
Out of device descriptors, increase gdt size
(NGDT) and relink XENIX
```

Otherwise, the routine returns the 16-bit selector number of the allocated descriptor.

Warning: It is important that the driver verifies that the return value is valid (not 0). Any attempt to use descriptor 0 may cause the kernel to panic.

Syntax:

dscrfree(sel)
unsigned short sel;

Description: The **dscrfree** routine returns a descriptor that is no longer needed to the pool of available device descriptors. It takes as its only argument the selector number returned from a call to **dscralloc**.

A device that uses a descriptor for most or all of its transfers should not release it, but should reuse the same descriptor for each transfer. Only devices that need a descriptor for a short period of time (during initialization, for example) should ever free a descriptor. Note that this routine can be used only with XENIX-286.

Parameters: The value of *sel* is an unsigned short value that specifies the selector number of the descriptor being freed.

Syntax:

paddr_t
dscraddr(sel)
unsigned short sel;

Description: The **dscraddr** routine returns the physical address of the memory addressed by the selector that is provided as the argument. Note that this routine can be used only with XENIX-286.

Parameters: The value of *sel* is an unsigned short value that specifies the selector number provided as the argument.

Return: The **dscraddr** routine returns *addr*, which is the 32-bit physical address of the memory addressed by the selector.

Syntax:
unsigned short ldtalloc (startsel, cnt)
unsigned short startsel;
int cnt;

Description: The **ldtalloc** routine allocates free user mapping descriptors ensuring that the last entry in the Local Descriptor Table (LDT) is always free. Note that this routine can be used only with XENIX-286.

Parameters: If *startsel* = 0, **ldtalloc** returns the first free selector (or block of free descriptors if *cnt* > 1). If *startsel* != 0, **ldtalloc** allocates the given selector (or block of selectors) if they are available for use. This function may only be used at task time.

Return: **ldtalloc** returns MMUERR ((unsigned) -1) on error; the number of the starting selector on success.

Syntax:
ldtfree(startsel,cnt)
unsigned short startsel;
int cnt;

Description: The **ldtfree**() routine frees user mapping descriptors that were previously allocated using **ldtalloc**(). This function may only be called at task time. Note that this routine can be used only with XENIX-286.

Parameters: *startsel* is the selector to be freed (or the starting selector of a block of selectors if cnt > 1). *cnt* is the number of sequential selectors to be freed (typically 1).

Syntax:

```
mmudescr (sel, addr, limit, access)
unsigned short sel;
paddr_t addr;
unsigned short limit;
char access;
```

Description: The **mmudescr** routine initializes a descriptor to map an area of memory.

Parameters: The value of *sel* is an unsigned short value that specifies the selector number of the descriptor allocated by **dscralloc**() or **ldtalloc**().

The value of *addr* is a long value that specifies the address of the beginning of the memory area to be mapped.

The value of *limit* is an unsigned short value that specifies the limit of the memory area (its size in bytes, minus 1).

The value of *access* is an unsigned char value that specifies an access designation. The possible values of access are RO, RW, and DSA_DATA (as defined in /usr/sys/h/mmu.h and /usr/sys/h/relsym86.h). Here is a table of the possible values of *access*:

RO	Specifies read-only access.
RW	Specifies read/write access.
DSA_DATA	Use for local driver selectors.

Both RO and RW allow user access to the selectors, DSA_DATA should be used for selectors that are internal to the driver.

Example: The **mmudescr** routine maps a section of memory 1024 bytes long at address 0xB0000 for reading and

writing as follows:

```
mmudescr(sel, 0xB0000, 0x3FF, RW);
```

When writing to a device driver that uses a selector to map memory, follow these steps:

1. Use **dscralloc**() in the driver initialization routine (or on first open for this device) to reserve a descriptor for the driver's use.

2. For each data transfer, use **mmudescr**() to set the descriptor to map the area of memory that the driver needs to access.

As an example, the following code allocates a descriptor, then maps a 512 byte area beginning at address 0xB0000 into the kernel's address space. Remember that the third argument is the limit of the transfer, not its size. Once the area has been mapped, the sotofar() macro may be used to convert the segment-offset pair to a far address (faddr) which can then be used like any other kernel logical address.

```
int seg;
faddr_t faddr;

if ((seg = dscralloc()) == 0)
{
      /* error processing */
      return;
}
mmudescr(seg, 0xB0000, 511, RW);
faddr = sotofar(seg, 0);
```

Syntax: **mloc_t mmuget(npage)**
msize_t npage;

Description: The **mmuget** routine may be used to allocate memory from the system memory map. Memory is managed and allocated in units of "pages" (one page = 512 bytes on an 80286). The base address returned by **mmuget**() may be used with **mmudescr**() to create a valid mapping, so that the memory can then be used by the driver. Note that this routine can be used only with XENIX-286.

Parameters: The argument *npage* is an unsigned integer which specifies the number of pages to allocate. The **btoms**() macro, defined in */usr/sys/h/sysmacros.h,* is provided to perform the bytes-to-pages conversion.

Return: **mmuget**() returns the page number of the first page in the allocated segment. This value may then be converted to a physical address suitable for use with **mmudescr**() - by using the **mltoa**() macro. **mmuget**() returns MMUERR on failure.

Example: This code allocates *nbytes* bytes of memory, placing the segment of the allocated read in the variable *seg*. Note that all three variables defined are used again when the memory is freed with **mmufree**().

```
int seg, nbytes;
unsigned short base;

#ifdef M_I286
  base = mmuget(btoms(nbytes));
  seg =  dscralloc();
  mmudescr(seg, mltoa(base), nbytes, RW)
#endif
```

Syntax: **mmufree(basepage, npage)**
mloc_t basepage;
msize_t npage;

Description: This routine deallocates memory previously allocated by **mmuget**(). Memory must be freed in "pages" - the same unit as in which it was allocated. Note that this routine can be used only with XENIX-286.

Parameters: This routine takes the base value (or starting page), which is the unsigned integer returned from an **mmuget**() call, and the number of pages to free.

Example: This code deallocates *nbytes* of memory, freeing the memory selector at the same time. The variables *seg*, *nbytes*, and *base* are taken from the previous **mmuget**() example.

```
int seg, nbytes;
unsigned short base;

#ifdef M_I286
  mmufree(base, btoms(nbytes));
  dscrfree(seg);
#endif
```

386-Specific Memory Management Routines

The following memory management routines are specific to XENIX-386:

Syntax: **IS386();**

Description: This macro is defined in */usr/include/sys/user.h*. It is used to find out if the program that called the driver is a 386 binary. This is required in a driver's ioctl function that receives an address. Addresses passed to the driver from an 8086 or 80286 binary are in segment/offset format, and must be converted to an 80386 virtual address using **cvttoaddr**() (see below). **IS386**() provides a way to determine if address conversion is necessary. Note that this routine can be used only with XENIX-386.

Example: The following code fragment demonstrates an ioctl function that uses **IS386**():

```
struct foo fip;

xxioctl( dev, cmd, arg )
dev_t dev;
int cmd;
faddr_t *arg;
{
      if( !IS386() )
      arg = (faddr_t)cvttoaddr( arg );
      copyout( &fip, arg, sizeof( struct foo ) );
}
```

Syntax: **caddr_t cvttoaddr (addr286)**
faddr_t addr286;

Description: The IS386() macro can be used by the driver to determine whether the calling process is an 80386 binary, or an 8086/80286 binary. The **cvttoaddr** routine is used in driver ioctl routines when dealing with 286 binaries. The routine converts a segmented program's

far address into an 80386 virtual address that may be used in an 80386 driver, such as to pass to **copyin**() or **copyout**(). The most common use for this conversion is to convert a segmented address passed to the driver though an **ioctl**() call. Addresses in the process's user structure are converted automatically by the kernel, so using **cvttoaddr**() is not necessary. Note that this routine can be used only with XENIX-386.

Parameters: The *addr286* argument is a 286 far pointer.

Return: The linear data address, as an offset from the beginning of the user's data space, is returned.

Syntax:
int **cvttoint (addr286)**
faddr_t addr286;

Description: The **cvttoint** routine is a portable way to extract the low 16-bit word from a 32-bit word.

Parameters: The *addr286* argument is a 286 far pointer.

Return: The low 16-bits of the 32 bit word are returned.

Syntax:
mapphys (physaddr, nbytes)
char *physaddr;
int nbytes;

Description: The **mapphys** routine maps physical memory addresses to virtual memory addresses for use by the kernel. This is necessary to allow a device driver to access some physical area of memory address space, such as a memory-mapped device, or RAM or ROM at a particular physical address. The value returned by mapphys may be used like any other character pointer. The second argument to mapphys must be the length of the area to access.

For example, to access 16K of video RAM on a CGA card which lies at physical address 0xB8000, use the following statement:

```
char *ptr;

ptr = (char *) mapphys( 0xB8000, 0x4000 );
```

Note that this routine can be used only with XENIX-386.

Parameters: The argument *physaddr* is the physical memory address to map.

The argument *nbytes* is the number of bytes to map, starting at *physaddr*.

Return: The virtual address that maps to the physical address *physaddr* is returned.

Syntax: **void unmapphys (va, nbytes)**
char *va;
int nbytes;

Description: The **unmapphys** function takes the virtual address returned by **mapphys**(), and unmaps it. The virtual address is then free for later use. The size should be the same as the size used in the original call to **mapphys**(). Once **unmapphys** has been called, accessing the virtual address without remapping will cause a trap inside the kernel. This routine must be used on addresses obtained with **mapphys**().

For example, to unmap the pointer allocated by the **mapphys** example, use the statement:

```
unmapphys( ptr, 0x4000 );
```

Note that this routine can be used only with XENIX-386.

Parameters: The argument *va* is the virtual address returned from a previous call to **mapphys**().

The argument *nbytes* is the number of bytes to free. This should be the same number of bytes that was allocated with **mapphys**().

Syntax: **mapptov (physaddr, vaddr, nbytes)**
char *physaddr;
unsigned int vaddr;
int nbytes;

Description: The **mapptov** routine maps physical memory addresses to specific virtual addresses.

Parameters: The argument *physaddr* is the physical memory address to map.

The argument *vaddr* is the virtual address to map.

The argument *nbytes* is the number of bytes to map, starting at *physaddr*.

Warning: This routine may only be called in the driver's initialization function.

Return: The virtual address that maps to *physaddr* is returned.

Syntax: **pfn_t**
memget (clicks)
pfn_t clicks;

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 "clicks." Use the macro **btoc()** to calculate the number of clicks from the number of bytes required. **memget()**'s return value is also in clicks, so the **ctob()** 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>*.

For example, to obtain a permanent 4K buffer for a driver, use the following code statement:

```
char *always;

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

Parameters: *clicks* is the number of clicks to allocate.

Warning: This routine may only be called in the driver's initialization function.

Return: The page frame number of the first frame of memory allocated is returned.

Syntax:

```
char*
sptalloc (nbytes)
unsigned int nbytes;
```

Description: The **sptalloc()** routine is used to obtain temporary memory for use by device drivers when more than one page of kernel-addressable memory is needed. This memory is obtained from the system's virtual memory pool. Because the virtual memory management is not set up until after initialization, **sptalloc()** must not be called during a driver's initialization function (use **memget()** instead). When the driver is through with the memory, the memory should be released via **sptfree()**. This routine returns a virtual address usable by any kernel or driver routine.

For example, to allocate a temporary 4K buffer, use the following code statement:

```
char *tmp;

tmp = sptalloc( 0x1000 );
```

Parameters: The argument *nbytes* is the number of bytes of contiguous kernel-addressable memory to allocate.

Warning: Note that **sptalloc()** may not be called until after driver initialization. Also, because **sptalloc()** may sleep, it should not be used at interrupt time.

Return: This routine returns the kernel virtual address of the memory allocated .

Syntax:
```
void sptfree (va, nbytes, freeflg)
char *va;
int freeflg;
```

Description: The **sptfree()** routine frees memory obtained from **sptalloc()**. 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, nbytes, 1 );
```

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

The value of *nbytes* is the number of bytes to free. This should be the same number of bytes that were allocated with **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 when another process will continue to use the memory (for example, the u-area is allocated in this way).

Chapter 7

Data Manipulation Routines

7.1 Data Manipulation Routines

This section describes routines that manipulate data and copy bytes to and from specific locations in the kernel and user space.

For 286 and 386: **copyio(), copyin(), copyout(), bcopy(), fubyte(), fuword(), subyte(), suword()**

For 386 only: **bzero(), clrbuf()**

Syntax:

int **copyio (addr, faddr, cnt, mapping)**
paddr_t addr;
faddr_t faddr;
unsigned cnt;
int mapping;

Description: The **copyio** routine copies bytes to and from a physical address (buffer address) in the kernel to and from a far (32 bit) address (user data pointer).

Parameters: The argument *addr* is a pointer to the physical kernel address to which or from which the data is to be transferred.

The argument *faddr* is a 32-bit pointer that contains the offset of the user address to which or from which the data is to be transferred.

The argument *cnt* is an unsigned 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 */usr/include/sys/user.h*:

Value	Definition
U_WUD	Transfers from user data to kernel data (buffer)
U_RUD	Transfers from kernel data (buffer) to user data

U_WKD	Transfers from kernel data to file (buffer)
U_RKD	Transfers from file (buffer) to kernel data

Return: If successful, this routine performs the specified data transfer; otherwise, it returns -1.

Example: For 286 only

The **ioctl** interface to a driver actually has two calling sequences:

```
1)    ioctl (fd, cmd, arg)
        int fd, cmd, arg;

2)    ioctl (fd, cmd, arg)
        int fd, cmd;
        char *arg;
```

In the kernel, the **ioctl** interface is translated into the device-specific call shown in the following example:

```
xxioctl (dev, cmd, arg)
int dev, cmd;
faddr_t arg;
```

If *arg* is a pointer to a data structure, copy your data in and out using the **copyio** routine as shown in the following example:

```
struct foo dst;
.
.  other ioctl code
.
/* copy from arg to dst */
if ( copyio (ktop(&dst), arg,
    sizeof(struct foo), U_WUD) == -1 ) {

      u.u_error = EFAULT;
      return;
}
```

Note

The <*sys/param.h*> and <*sys/sysmacros.h*> files define several useful macros for converting 286 addresses from one type to another. These macros include the macros in the following list:

Macro	Function
ftoseg(x)	Converts *x* from a *faddr_t* to a 16-bit segment (selector) number.
ftooff(x)	Converts *x* from a *faddr_t* to an offset.
sotofar(seg,off)	Converts a segment, offset pair into a *faddr_t* .
ptok(x)	Converts a physical address to a kernel logical address.
ktop(x)	Converts a kernel logical address to a physical address.

Syntax: **bcopy (src, dst, cnt)**
char *src, *dst;
int cnt;

Description: The **bcopy** routine copies bytes in kernel space.

Parameters: 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.

The value of *cnt* is the number of bytes to transfer.

Syntax: **copyin (src, dst, cnt)**
faddr_t src;
char *dst;
int cnt;

Description: The **copyin** routine copies bytes from the user's data space to the kernel's data space.

Parameters: 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.

Return: If successful, this routine performs the specified data transfer; otherwise, it returns -1.

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)
int dev, cmd;
faddr_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;
        }
```

Syntax:

```
copyout (src, dst, cnt)
char *src;
faddr_t *dst;
int cnt;
```

Description: The **copyout** routine copies bytes from the kernel's data space to the user's data space.

Parameters: 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: If successful, this routine performs the specified data transfer; otherwise, it returns -1.

Example: Assuming *arg* is a pointer to a user data structure that was passed via **xxioctl()**, use **copyout()** to copy from kernel data space to user data space.

```
xxioctl(dev, cmd, arg)
int dev, cmd;
faddr_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;
      }
.
.
```

386-Specific Data Manipulation Routines

Syntax: **bzero (p, cnt)**
char *p;
int cnt;

Description: The **bzero** routine sets memory locations to 0.

Parameters: The argument *p* specifies the beginning of the area to clear.

The value of *cnt* is the number of bytes to set to 0.

Syntax: **fubyte (src)**
faddr_t src;

Description: The **fubyte** routine retrieves (fetches) one character from the user's data space.

Parameters: The argument *src* is a 32-bit pointer that contains the offset of the user address the character is copied from.

Return: The value of the retrieved byte is returned.

Syntax: **fuword (src)**
faddr_t *src;

Description: The **fuword** routine retrieves (fetches) one word from the user's data space.

Parameters: The argument *src* is a 32-bit pointer that contains the offset of the user address the word is copied from.

Return: The value of the retrieved word is returned.

Syntax: **subyte (addr, val)**
char *addr;
int val;

Description: The **subyte** routine sets one character in the user's data space.

Parameters: The argument *addr* is a pointer to the byte to be set in the user's data space.

The argument *val* is the value to be set.

Syntax: **suword (addr, val)**
char *addr;
int val;

Description: The **suword** routine sets one word (four bytes) in the user's data space.

Parameters: The argument *addr* is a pointer to the beginning of the four bytes to be set in the user's data space.

The argument *val* is the value to be set.

Chapter 8

Direct Memory Allocation Routines

8.1 DMA Routines

The functions in this section interface to the Direct Memory Access (DMA) controller. After the discussion on using the DMA functions, there is a section on the syntax and purpose of each DMA function.

8.1.1 Using the DMA Support Functions

There are two possible methods of performing a DMA transfer in the XENIX kernel. The first method uses the queue of DMA requests maintained by the kernel. The second method is for the driver itself to allocate the DMA channel, start the transfer, check to see that the transfer was completed, then release the channel. This can be done in the task time portion of a driver. However, because this process can take a substantial amount of time (especially if there is a long wait for the DMA channel to become available), it is not suitable to do this at interrupt time. The first method is better suited for interrupt time, or for drivers that must share a DMA channel with other devices. This method allows a driver to submit a request to a queue of DMA requests maintained by the kernel, then continue with other processing. The kernel processes each request in turn.

The steps required to perform a DMA transfer are as follows: **dma_alloc()** 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()** with the appropriate parameters. After the parameters have been set, the driver begins the actual transfer by calling **dma_enable()**. After the transfer is complete, the function **dma_resid()** may be called to find out the amount of data that was not transferred. A condition where data was not transmitted can be caused by a DMA request that crosses a segment boundary. Another DMA transfer must be initiated to complete the request. Once the transfer is completed, the driver must call **dma_relse()** to release the channel for use by other drivers.

The following function demonstrates the use of the DMA functions. It accepts a buffer and a byte count, and writes the data in the buffer to DMA channel 1:


```
#define  FREDS_CHANNEL   1

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

  if ( dma_alloc( FREDS_CHANNEL, DMA_BLOCK ) ) {
```

```
    printf( "Error: couldn't allocate DMA channel" )
  else {
    dma_param(FREDS_CHANNEL, 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 */

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

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

Using the kernel's DMA request queue requires a slightly different procedure than given above. To submit a request, the driver function calls **dma_start()**, passing to it a dmareq structure defining the DMA request. The driver's function 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 function to do the transfer
d_param: parameter to the function pointed to by d_proc

The d_proc element should point to the function to be called by the kernel when it is time to service this particular request. This function 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 function should be as short as possible, since it may be called during another driver's interrupt function. When this service function 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 function must call **dma_relse()** to release the DMA channel.

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

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

```
#define  MY_CHANNEL  1

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

  /* set up DMA request structure */

  dp.d_chan = MY_CHANNEL;
  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 requests 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 )
printf( "Error: DMA not completed,");
printf("%ld bytes not transferred\n", leftover );
  return(0);
}


service_dma( dp )
struct dmareq *dp;
{
  printf( "Now servicing %s0, 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 function that waits for signal from device */

  leftover = dma_resid();
  if ( leftover > 0L )
    printf( "Error: %s not completed,
           %ld bytes not transferred0,
```

```
            dp->d_params, leftover );
  dma_relse( dp->d_chan );
  /* wake up sleeping requestor, if any */
  wakeup( &leftover );
  return(0);
}
```

The functions in this section interface with the Direct Memory Access (DMA) controller. These are available for both the 286 and the 386.

Syntax: **dma_alloc (chan, mod)**
unsigned chan, mod;

Description: The **dma_alloc** function allows dynamic allocation of a DMA channel. You should not use both dma_alloc() and dma_start() at the same time.

Parameters: The *chan* argument specifies the channel to be allocated.

The *mod* argument can have one of two values:

DMA_BLOCK	Waits until the channel is available.
DMA_NBLOCK	Returns immediately with a return stat 0 if the channel was not free at this tim

If *mod* specifies blocking, the **dma_alloc** function 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** function 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.

This function should not be used in immediate conjunction with dma_start(). Make certain that your DMA channel has been allocated before beginning your operations. An example of how to use this function is:

```
/* allocate  channel 1. */
/* If not currently available, wait */

if( dma_alloc( 1, DMA_BLOCK ) == 0 )
```

```
{
        seterror( EIO );
        return;
}

/* If channel is successfully allocated, */
/* then begin DMA streaming */

dma_start( dma_request )
struct dmareq {
        struct dmareq       *d_nxt;
        unsigned short      d_chan;
        unsigned short      d_mode;
        paddr_t         d_addr;
        long            d_cnt;
        int             (*d_proc)();
        char            d_params;
} *dma_request;
```

This function sets up the XENIX 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 d_proc with a pointer to the function to be called once the dma channel is allocated, the driver can let the kernel handle the allocation of the DMA channel.

When the channel is allocated, the function pointed to by d_proc will be called with a pointer to dma_request. At this point, the dma channel has been allocated as if the driver had done so with **dma_alloc()**.

If the function was not able to allocate the channel immediately, but had to queue your request, this function will return a 0.

For example, allocating DMA channel 1 with modes of DMA_BLOCK, and we want foo_proc() to be called when we have the channel allocated:

```
/* set up dma structure */
extern int foo_proc();

struct dmareq foo_req = {
        NULL,
        1,
        DMA_Rdmode,
        (paddr_t)0,
        0L,
```

```
        foo_proc,
        0,
};

        .
        .
        .
        dma_start( &foo_req );
        /* we don't care if we are queued or not */
        return;
```

Syntax: **dma_start (arg)**
struct dmareq *arg;

Description: The **dma_start** function requests a DMA transfer to begin. Interrupt functions can use this facility. The format of the *dmareq* structure is as follows:

```
struct dmareq {
  struct dmareq *d_nxt;
  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 function 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 function to call when the channel is available, and an address of further data that may be needed by the **d_proc** function.

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** with a pointer to *arg* as a parameter. The **dma_start** function 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() simply returns 0.

Note: The **d_proc** functions will be executed at **spl6** and should observe all the normal rules of the MP interrupt functions. 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**, **delay**, or other functions that may cause **sleep** to be called cannot be used.

Syntax:
dma_relse (chan)
short chan;

Description: The **dma_relse** function releases a DMA channel previously allocated with **dma_alloc()** or **dma_start()**. This function 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 function has no return value. If you intend to share DMA channels, you should use this function. 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 **d_proc** function with a pointer to the *dmareq* structure as a parameter. It is clear that because **d_proc** may be called during another driver's interrupt, it should be as minimal as possible to accomplish its task.

Parameters: The argument *chan* is the DMA channel to be released.

Example: To release the channel that was allocated in the previous allocation examples:

```
/* finished with DMA for now, release channel */
        dma_relse( 1 );
```

The following DMA functions are to be used by **d_proc** functions or task time code after the **dma_alloc** function has been used to successfully allocate a DMA channel:

Syntax: **dma_param (chan, mode, addr, cnt)**
unsigned chan, mode;
paddr_t addr;
long cnt;

Description: This function will set up the controller chip for your DMA transfer. The **dma_param()** function masks the DMA request line on the DMA controller, sets the address and count parameters, and sets the mode (read or write). This must always be used once the dma channel has been allocated for the driver by the functions **dma_alloc()** or **dma_start()**. In the case of the driver using **dma_start()**, this would be called by the function pointed to by *d_proc*. There is no return value.

Parameters: The *chan* argument specifies the DMA channel to be used.

The *mode* argument specifies whether this is a read or write transfer. The options are:

DMA_WRMODE (0x48). This option specifies a transfer from memory to a device.

DMA_RDMODE (0x44). This option specifies a transfer from a device to memory.

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 );
.
.
}
```

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.

Syntax: **dma_enable (chan)**
unsigned chan;

Description: This function starts the DMA transfer set up by dma_param(). There is no return value. This function clears the mask register on the controller to let the DMA transfer begin.

Parameters: The *chan* argument specifies the DMA channel to be used.

Syntax: **long dma_resid (chan)**
unsigned chan;

Description: This function returns the number of bytes not transferred by the DMA request as a long integer.

Parameters: The *chan* argument specifies the DMA channel to be queried.

Return: The **dma_resid** function returns the number of bytes that were not transferred.

Chapter 9

Kernel Support Routines

9.1 Kernel-Support Routines

This chapter describes the routines (interfaces) that the kernel provides to facilitate developing device-drivers. With SCO XENIX System V, each device driver can call only the routines described in this chapter. For more information on calling functions, see the *C Language Reference*.

In XENIX-286, device drivers must be compiled as medium model programs. This is taken care of automatically by the C compiler when you use the **-Mm** option to specify that the driver is to be compiled using the Medium size memory model.

9.1.1 Input/Output Routines

This section describes the routines that allow access to device registers in I/O space.

For 286 and 386: **inb(), outb(),**

For 286 only: **in(), out()**

For 386 only: **inw(), ind(), outw(), outd(), repinsb(), repinsd(), repinsw(), repoutsb(), repoutsd(), repoutsw()**

Note that the various **out** and **rep** instructions do not return specific values. Do not attempt to use their return values as a means of error checking as these values are not consistent from release to release.

The following two routines, inb() and outb(), provide a portable interface to the i/o space addresses on your device controller or adapter, These two routines can be used with both XENIX-286 and XENIX-386.

Syntax: **int inb (read_addr)**
int read_addr

Description: The **inb** routine reads a byte from the I/O address specified by the parameter read_addr. This routine is available on both XENIX-286 and XENIX-386.

Parameters: The value of *read_addr* is an integer that specifies the physical I/O address that is to be read.

Return: The value of the byte located at the 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);
```

Syntax: **outb (write_addr, value)**
int write_addr;
char value;

Description: The **outb** routine writes the byte specified by *value* to the physical I/O address specified by *write_addr*. This routine is available on both XENIX-286 and XENIX-386.

Parameters: *write_addr* is an integer that specifies the physical I/O address that will be written to.

value is the byte that will be written to the physical I/O address *write_addr*.

Example: 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);
```

286-Specific Routines

The following I/O routines are specific to XENIX-286:

Syntax: **in (read_addr)**
int read_addr;

Description: The **in** function reads a 16-bit word from the physical I/O address specified by read_addr. This routine is only available on XENIX-286.

Parameters: *read_addr* is an integer that specifies the physical I/O address being read from.

Return: The value of the 16 bit word at the I/O address specified by *read_addr* is returned.

Example: To read the status of a word register at I/O address 0x20, use the following lines of code:

```
int   val;
val = in(0x20);
```

Syntax: **out(write_addr, value)**
int write_addr, value;

Description: The **out** function writes the 16-bit integer specified by *value* to the physical I/O address specified by *write_addr*. This routine is only available on XENIX-286.

Parameters: *write_addr* is the physical I/O address being written to. *value* is the 16-bit integer that will be written.

Example: To write the 16-bit value 0xff0 to I/O address 0x300 you could use the following line of code:

```
out(0x300,  0xff0);
```

386-Specific Routines

The following I/O routines are specific to XENIX-386:

Syntax: **int inw (read_addr)**
int read_addr;

Description: The **inw** function reads a 16-bit word from the physical I/O address specified by read_addr. This routine is only available on XENIX-386.

Parameters: *read_addr* is an integer that specifies the physical I/O address to be read from.

Return: A 32-bit integer whose high 2 bytes are set to zero is returned.

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);
```

Syntax: **int ind (read_addr)**
int read_addr;

Description: The **ind** function reads a 32-bit word from the physical I/O address specified by read_addr. This routine is only available on XENIX-386.

Parameters: The value of *read_addr* is an integer that specifies the physical I/O address to be read from.

Return: **ind** returns the 32-bit value read from the I/O address *read_addr*.

Example: To read a 32-bit value from I/O address 0x300 you could use the following code:

```
int   val;
val = ind(0x300);
```

Syntax: **outw (write_addr, value)**
int write_addr, value;

Description: **outw** writes a 16-bit word to the physical I/O address specified by *write_addr*. This routine is only available on XENIX-386.

Parameters: *write_addr* is the physical I/O address being written to. *value* is the 16-bit word bring written to *write_addr*.

Example: To write a the 16-bit value 0xff0 to I/O address 0x300 you could use the following code:

```
outw(0x300, 0xff0);
```

Syntax: **outd (write_addr, value)**
int write_addr, value;

Description: **outd** writes a 32-bit value to the physical I/O address specified by *write_addr*. This routine is only available on XENIX-386.

Parameters: *write_addr* is the physical I/O address being written to. *value* is the 32-bit word being written to *write_addr*.

Example: To write the 32-bit value 0xffff00 to I/O address 0x300 you could use the following code:

```
outd(0x300, 0xffff00);
```

Syntax: **repinsb (dev_addr, kv_addr, cnt)**
int dev_addr, cnt;
caddr_t kv_addr;

repinsw (dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;

repinsd (dev_addr, kv_addr, cnt)

int dev_addr, cnt;
caddr_t kv_addr;

Description: The **repin** functions 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 functions are typically used for reading from disk and SCSI devices. These functions assume that the virtual address specified is a valid kernel address currently in RAM. These routines is only available on XENIX-386.

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.

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()**

Examples: The following three examples demonstrate how to use the **repin** functions. 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);
```

Syntax:

repoutsb (dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;

repoutsw (dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;

repoutsd (dev_addr, kv_addr, cnt)
int dev_addr, cnt;
caddr_t kv_addr;

Description: The **repout** functions 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 functions are typically used for writing to disk and SCSI devices. These functions assume that the virtual address specified is a valid kernel address currently in RAM. These routines are only available on XENIX-386.

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.

Parameters:

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 **repout** functions. 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);
```

9.1.2 Interrupt Support Routines

This section describes the routines used to enable and disable interrupts during task-time processing. These routines alter the system priority level (spl) and should be used only to protect critical sections of your code. They can also be used to guarantee uninterrupted execution of device driver code. However extreme care should be taken to avoid spending long periods of time (hundreds of microseconds) above spl 5 as this will cause the software clock to lose time and will reduce performance. Please note that the the functions spl1(), spl2(), spl3(), spl4(), and spl6() are also provided, although they are not used by the XENIX kernel to protect specific data structures. Their use is left to the discretion of the device driver writer. All the spl functions except for splx() return the previous spl.

splcli(), splx(), spl0(), spl5(), spl7()

Syntax: **splx (oldspl)**
int oldspl;

Description: The **splx** routine sets the system priority level to the level specified by *oldspl*, *oldspl* should have been set to the return value of a previous call to one of the other spl functions such as: **splcli() , spl5() , spl6() ,** or **spl7()**. Calls to any one of these routines and the **splx** routine nest correctly.

Parameters: The integer *oldspl* specifies a previous spl level, it should only be set by the return value of a previous spl function.

Example: See the example following the discussion of **splcli()**.

Syntax: **int splcli ()**

Description: This routine sets the equivalent of spl 5, it should be used instead of the spl5() routine for compatibility. It should be used to protect critical sections of code which manipulate clist structures or pointers. It is possible that a device driver's xxpoll function will preempt another driver while it is manipulating clists. If your xxpoll function manipulates clist structures you should exercise care to make sure that your function was not entered at an spl level higher than 5. Otherwise you may corrupt the kernel free clist. You should not manipulate clists in your xxintr function.

Return: The previous spl value is returned. This value may be used to restore interrupts with the **splx** routine.

Note: It is not necessary to use splcli() before calling any of the cblock or clist functions supplied with XENIX (getcb(), getcf(), putcb(), etc) because these functions will 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.

Example: Here are some code fragments which show how to check spl level in an xxpoll function and to demonstrate the use of splcli() in a task time function.

```
/*
 * This macro tells us if the previous spl level was "lev"
 * or higher. PS_PRIMASK is defined in <sys/param.h>.
 */
#ifdef M_I386
#define ATSPL(lev,ps)  ((ps) >= lev)
#else
#define ATSPL(lev,ps)  (((ps&PS_PRIMASK)>>8) >= lev)
#endif

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 example
 * of task time functions.
 */
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.
*/
}
```

Syntax: **int spl0()**

Description: The **spl0** functions sets the lowest spl possible. It is used to assure that no device interrupts are being blocked.

Return: The previous spl value is returned. It may be saved and used to restore the spl level by a subsequent call to **splx()**.

Syntax: **int spl5 ()**

Description: The **spl5** routine typically masks all interrupts except for those from the clock and serial devices. This routine is provided for backward compatibility, so if you are writing a serial device driver, you should use **splcli()** whenever possible.

Return: The previous splvalue is returned. It may be saved and used to restore the spl level by a subsequent call to **splx()**.

Syntax: **int spl6 ()**

Description: The **spl6** routine typically masks all interrupts except for those from the serial device. In general there is no good reason for setting an spl of 6, it should only be done in extreme circumstances where it is necessary to assure that a given section of code will execute without being pre-empted. Typically this is only necessary where some semblance of real time response is necessary.

Note: Use of this function causes 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.

Return: The previous spl value is returned. It may be saved and used to restore the spl level by a subsequent call

to **splx()**.

Syntax: **int spl7 ()**

Description: The **spl7** routine disables all interrupts. Use this routine only for extremely short periods when updating critical data structures that could be accessed by a high priority device.

Note: This function causes 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. In addition, since this priority level blocks all interrupts, characters will not be echoed back to the console, and the capslock, numlock, and scrolllock indicators will not work. Additionally, it is not possible to switch multi-screens while a driver is at spl 7. If for some reason your device driver becomes hung at spl 7 the system will be frozen and your only option will be to power cycle the machine.

Return: The previous spl value is returned. It may be saved and used to restore the spl level by a subsequent call to **splx()**.

9.1.3 Timing and Synchronization Functions

This section describes the functions **sleep()**, and **wakeup()**, which may be used to suspend driver execution until a shared resource becomes available, and the functions timeout()**, and delay()**, which may be used to temporarily halt task time processing in a driver or schedule the execution of a function at a specified time in the future. These functions change the spl to zero and cause a context switch to occur. They should never be used in driver interrupt handling code. After a sleep has been awakened or a timeout or delay has expired, control will return to the driver's task time function. All of these functions return the system to the priority level that was in effect before they were executed.

The **timeout()** and **delay()** functions use a constant size data structure. If this data structure overflows, the error:

```
panic: timeout table overflow
```

```
/* reputsb */
char kv_addr[50];
reputsb(dev_addr, (caddr_t) kv_addr, 50);

/* reputsw */
short int kv_addr[50];
reputsw(dev_addr, (caddr_t) kv_addr, 50);

/* reputsd */
int kv_addr[50];
reputsd(dev_addr, (caddr_t) kv_addr, 50);
```

occurs. If this happens, it is necessary to reconfigure the kernel and increase the value of the NCALLS parameter.

The following functions are described: **sleep(), wakeup(), timeout(), delay()**

Syntax: **sleep (wait_channel, priority)**
caddr_t wait_channel;
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()** 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. This routine should never be called at interrupt time.

Parameters: *wait_channel* 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. During debugging it is useful for the device driver writer to display (see Miscellaneous Support Routines) 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 will place 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 (include <sys/param.h>) 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 will be able to force termination of their processes by a software signal if an error has occurred.

Return: **sleep**() returns 0 if a **wakeup**() function has been called using the same wait_channel that was specified in the **sleep**() call, or it returns 1 if the priority used has been or'd with PCATCH (include <sys/param.h>) and the sleeping process has been sent a software signal.

Note: If *priority* has not been or'd 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**() 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 if the **sleep**() is interrupted, it is necessary that PCATCH be or'd into *priority*. This will cause control to return to the driver on a software interrupt. 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**() function for more information.

Syntax:

```
wakeup (wait_channel)
caddr_t wait_channel;
```

Description: **wakeup** causes all processes which are sleeping at a wait channel equal to *wait_channel* to be taken off the sleep queue and placed on the run queue. When a process is awakened, the call to **sleep**() 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: *wait_channel* should be the same value used in a previous invocation of the **sleep()** function. Since this number is not guaranteed to be unique and multiple processes may have have been awakened by any single invocation of the **wakeup()** function it is not guaranteed that the event being waited for has in fact occurred.

Return: **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()* function allocates a temporary storage area, queues an I/O transfer, and then puts the process to sleep. The *xxintr()* function is called when the device is ready to do the transfer. After the transfer is complete the *xxintr()* function executes a *wakeup()*.

```
/*
 * declare variable which is used for wait_channel
 */
char my_wait_channel;

/*
 * 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 will only indicate transfer is done if wakeup() has
```

```
         * been called by my xxintr().
         */
        while (/* flag indicates transfer is not done */) {
        /* or MYPRI with PCATCH because I need to clean things up */
        if (sleep(&my_wait_channel, 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_wait_channel);
}
/*
 * Note that the preceding example does not take into
 * account the possibility that multiple user processes
 * may have queued requests for a single device.
 */
        /* 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);
```

Syntax: **timeout (function, arg, clock_ticks)**
int (*function) ();
caddr_t arg;
int clock_ticks;

Description: **timeout** schedules a function to be executed at a specific time in the future.

Parameters: *function* is the label for the function to be executed after the specified number of *clock_ticks* has elapsed.

arg is passed as a parameter to *function.*

clock_ticks is the number of clock ticks to wait before calling *function.* On most 286 or 386 machines a clock tick occurs 50 times per second (50Hz).

Note: **timeout()** should normally only be used at task time, however it can be used in an *xxinit()* function with the following warning: Since the clock interrupts may not be enabled before your *xxinit()* function is called, the timeout may not elapse when specified, but will elapse no later than (time_when_clock_started + clock_ticks) times one fiftieth of a second.

Example: **timeout()**, **sleep()** and **wakeup()** can be combined to provide a "busy, wait" function. The following code sample illustrates this possible functionality:

```
#define PERIOD  5 /* 5/50 equals 1/10 second */
#define BUSYPRI  (PZERO -1) /* arbitrary */

/* Declare function I will 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);
}
```

Note

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).

Syntax: **delay (ticks)**
int ticks;

Description: The **delay** routine uses **sleep**() and **wakeup**() calls to delay the current process for the specified number of clock ticks. Its functionality is similiar to that of the *sleep()* system service except that time is measured in fiftieths of a second. This function should not be used at interrupt time.

Parameters: *ticks* is an integer that specifies the number of clock ticks to delay. One clock tick equals 1/50th of a second.

Return: After the specified time, the delayed function resumes running. No value is returned.

Warning: The **delay** routine should not be called at device-initialization (*init*) time as the amount of time delayed will not be what is expected.

9.1.4 Process Control Functions

setjmp(), longjmp(), psignal(), signal()

Syntax: **setjmp (u.u_qsav)**
label_t u.u_qsav;

Note: Typically, the device driver writer should never invoke **setjmp()** as it will interfere with the Operating System's ability to recover from an interrupted system call. The kernel always executes a **setjmp(u.u_qsav)** after the switch to system mode after a *call sys* in the user process. Typically it is only used if a **sleep()** is interrupted by a software signal and PCATCH was not set. In this case **sleep()** executes a **longjmp(u.u_qsav)** back to the system call interface.

Description: **setjmp** saves a stack environment that can subsequently be restored using **longjmp()**. Used together this way, the **setjmp** and **longjmp** routines provide a way to execute a nonlocal goto. This routine has the same functionality as **setjmp(S)**.

A call to **setjmp(u.u_qsav)** causes the current stack environment to be saved in *u.u_qsav*. A subsequent call to **longjmp(u.u_qsav)** restores the saved environment and returns control to the point just after the corresponding **setjmp** call. The values of all variables accessible to the routine receiving control contain the values they had when **longjmp** was called.

Parameters: The *u.u_qsav* structure saves the current context.

Return: The **setjmp** routine returns 0 after saving the stack environment. If **setjmp** returns as a result of a **longjmp** call, it returns 1. There is no error return.

Syntax: **longjmp (u.u_qsav)**
label_t u.u_qsav;

Description: The **longjmp** routine restores the process context that was previously saved using **setjmp()**. This routine has the same functionality as **longjmp(S)**.

Parameters: *u.u_qsav* contains the information saved by the previous **setjmp**(). This is the only argument that should ever be passed to **longjmp**().

Syntax: **psignal (proc_ptr, sig)**
register struct proc *proc_ptr;
register int sig;

Description: **psignal** sends the specified signal *sig* (see <sys/signal.h>) to the process specified by *proc_ptr.*

Parameters: *proc_ptr* 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.

Syntax: **signal (pgrp, signum)**
int pgrp;
int signum;

Description: The **signal** routine sends the specified signal, *signum,* to all processes in the process group identified by *pgrp.*

Parameters: *pgrp* is an integer that specifies the process group number. At task time it 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.

signum is an integer that specifies the signal to be sent.

9.1.5 Miscellaneous Support Routines

This section describes miscellaneous kernel support routines.

panic(), printf(), putchar(), getchar(), suser()

Syntax: **panic (s)**
char * s;

Description: The **panic** routine takes a parameter **s** 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 *s* is an address of a string that describes the reason for the system failure.

Example: panic("the cpu has melted down");

Syntax: **printf (format, p1, p2, ...)**
char * format;

Description: The kernel **printf** routine prints error messages and debugging information on the system console. It is a simplified version of the standard C library **printf** routine. The special format characters understood by the kernel **printf**() are %s, %c, %d, %ld, %lx, %u, %D, %X, %x, and %o, as well as the NEWLINE (\n) and RETURN (\r) characters.

Parameters: The **printf**() format string is similar to the format parameter used by **printf(S)**, it is used to describe the additional parameters to be printed by the routine are *p1,p2,....*

Notes: This routine is not interrupt-driven and will therefore suspend all other system activities while it is executing.

This routine is similar to standard C library function **printf**(), except that only the formats specified here are valid, and precision is not supported.

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)
int dev, cmd;
faddr_t arg;
{
printf("dev == %d, cmd == %d, addr of arg == %x0, dev, cmd, arg);
    .
    .
    .
}
```

Syntax: **putchar (c)**
char c;

Description: The **putchar** routine is used by the printf() and panic() functions. This routine puts one character on the console, doing a "busy wait" rather than depending on interrupts.

Parameters: *c.* is the character that is printed on the console.

Syntax: **int**
getchar ()

Description: **getchar()** Can be used to temporarily halt execution of the kernel, and get input from a user.

Return: **getchar()** returns the character typed at the keyboard.

Example:

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

Syntax: **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.

Chapter 10

Example Driver Code

10.1 Introduction

This chapter provides code fragments from example drivers for line printers, terminals, and hard disk drives. Each segment of code is followed by general comments that describe the routines and explain key lines in the program.

Note that these are not complete working drivers and should not be expected to be comprehensive. Real working device drivers are long, complex programs. 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 */usr/sys/io* directory 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 the examples given in */usr/sys/io*.

10.2 Code Fragments from a Line Printer Driver

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

```
 1  /*
 2  ** lp- prototype line printer driver
 3  */
 4  #include "../h/param.h"
 5  #include "../h/dir.h"
 6  #include "../h/a.out.h"
 7  #include "../h/user.h"
 8  #include "../h/file.h"
 9  #include "../h/tty.h"
10  #include "../h/conf.h"
11
12
13  #define LPPRI      PZERO+5
14  #define LOWAT      50
15  #define HIWAT      150
16
17  /* register definitions */
18
19  #define RBASE      0x00         /* base address of registers */
20  #define RDATA      (RBASE + 0)  /* place character here */
21  #define RSTATUS    (RBASE + 1)  /* non zero means busy */
22  #define RCNTRL     (RBASE + 2)  /* write control here */
23
```

```
24   /* control definitions */
25   #define CINIT            0x01  /* initialize the interface */
26   #define CIENABL          0x02  /* +Interrupt enable */
27
28   /* flags definitions */
29   #define FIRST            0x01
30   #define ASLEEP           0x02
31   #define ACTIVE           0x04
32
33   struct clist lp_queue;
34   int lp_flags = 0;
35
36   int lpopen(), lpclose(), lpwrite(), lpintr();
37
38   };
```

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 56 to 65

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

```
56
57  lpopen(dev)
58  int dev;
59  {
60        if ( (lp_flags & FIRST) = = 0 ) {
61              lp_flags |= FIRST;
62              outb(RCNTRL, CRESET);
63        }
64        outb(RCNTRL, CIENABL);
65  }
```

The code is defined as follows:

Line no.	Definition
60-62:	If this is the first time (since XENIX was booted) that the device has been touched, the interface is initialized by setting the CRESET bit in the control register.
64:	Interrupts from this device are enabled by setting the IENABL bit in the control register.

lpclose: Lines 66 to 70

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

```
66
67  lpclose(dev)
68  int dev;
69  {
70  }
```

lpwrite: Lines 71 to 90

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

```
71  lpwrite(dev)
72  int dev;
73  {
74        register int c;
75        int x;
76
77        while ( (c = cpass()) >= 0 ) {
78              x = splcli();
79              while ( lp_queue.c_cc > HIWAT ) {
80                    lpstart();
81                    lpflags |= ASLEEP;
82                    sleep(&lp_queue, LPPRI);
83              }
84              splx(x);
85              putc(c, &lp_queue);
86        }
87        x = splcli();
88        lpstart();
89        splx(x);
90  }
```

The code is defined as follows:

Line no.	Definition
77:	While there are still characters to be transferred, do what follows.
78-85:	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.
87-88:	Make sure the printer is running by locking out interrupts and calling **lpstart**.

lpstart: Lines 91 to 98

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

```
91
92    lpstart
93    {
94          if ( lp_flags & ACTIVE )
95                return; /* interrupt chain is keeping printer going */
96          lp_flags |= ACTIVE;
97          lpintr(0);
98    }
```

The code is defined as follows:

Line no.	Definition
94-97:	If the printer is running, just return; otherwise, set ACTIVE, and call **lpintr** to start the transfer of characters.

lpintr: Lines 99 to 122

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

```
 99
100
101   lpintr(vec)
102   int vec;
103   {
104         int tmp;
105
106         if ( (lp_flags & ACTIVE) = = 0 )
107               return;            /* ignore spurious interrupt */
108
109         /* pass chars until busy */
110         while ( inb(RSTATUS) = = 0 && (tmp = getc(&lp_queue)) >= 0)
111               outb(RDATA, tmp);
112
113         /* wakeup the writer if necessary */
114         if ( lp_queue.c_cc < LOWAT && lp_flags & ASLEEP ) {
115               lp_flags &= ~ASLEEP;
116               wakeup(&lp_queue);
117   }
```

```
118
119         /* wakeup writer if waiting for drain */
120         if ( lp_queue.c_cc <= 0 )
121               lp_flags &= ~ACTIVE;
122   }
```

The code is defined as follows:

Line no.	Definition
106-107:	If **lpintr** is called unexpectedly, or the driver does not have anything to do, it just returns.
110-111:	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** and pass to the interface by writing to the data register.
114-116:	If the buffer has fewer than LOWAT characters in it and some process is asleep waiting for room, wake it up.
120-121:	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.

10.3 Terminal Driver Code Examples

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

```
 1   /*
 2   ** td- terminal device driver
 3   */
 4   #include "../h/param.h"
 5   #include "../h/dir.h"
 6   #include "../h/user.h"
 7   #include "../h/file.h"
 8   #include "../h/tty.h"
 9   #include "../h/conf.h"
10
11
12   /* registers */
13   #define RRDATA     0x01  /* received data */
14   #define RTDATA     0x02  /* transmitted data */
15   #define RSTATUS    0x03  /* status */
16   #define RCNTRL     0x04  /* control */
17   #define RIENABL    0x05  /* interrupt enable */
18   #define RSPEED           0x06  /* data rate */
19   #define RIIR       0x07  /* interrupt identification */
```

```
20
21   /* status register bits */
22   #define SRRDY              0x01  /* received data ready */
23   #define STRDY              0x02  /* transmitter ready */
24   #define SOERR              0x04  /* received data overrun */
25   #define SPERR              0x08  /* received data parity error */
26   #define SFERR              0x10  /* received data framing error */
27   #define SDSR         0x20  /* status of dsr (cd)*/
28   #define SCTS         0x40  /* status of clear to send */
29
30   /* control register */
31   #define CBITS5             0x00  /* five bit chars */
32   #define CBITS6             0x01  /* six bit chars */
33   #define CBITS7             0x02  /* seven bit chars */
34   #define CBITS8             0x03  /* eight bit chars */
35   #define CDTR         0x04  /* data terminal ready */
36   #define CRTS         0x08  /* request to send */
37   #define CSTOP2             0x10  /* two stop bits */
38   #define CPARITY            0x20  /* parity on */
39   #define CEVEN              0x40  /* even parity otherwise odd */
40   #define CBREAK             0x80  /* set xmitter to space */
41
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 NTDEVS             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_speedr: 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 102 to 144

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

```
102
103
104   tdopen(dev, flag)
105   int dev, flag;
106   {
107         register struct tty *tp;
108         int addr;
109         tdproc;
110         int   x;
111
112         if ( UNMODEM(dev) >= NTDEVS ) {
113               seterror(ENXIO);
114               return;
115         }
116         tp = &td_tty[UNMODEM(dev)];
117         addr = td_addr[UNMODEM(dev)];
118         if( (tp->t_lflag & XCLUDE) && !suser ) {
119               seterror(EBUSY);
```

```
120                 return;
121           }
122           if ((tp->t_state&(ISOPEN|WOPEN)) = = 0) {
123                 ttinit(tp);
124                 tp->t_proc = tdproc;
125                 tp->t_oflag = OPOST|ONLCR;
126                 tp->t_iflag = ICNTRL|ISTRIP|IXON;
127                 tp->t_lflag = ECHO|ICANON|ISIG|ECHOE|ECHOK;
128                 tdparam(dev);
129           }
130           x = splcli();
131           if ( ISMODEM(dev) ||
132           tp->t_cflag & CLOCAL ||
133           tdmodem(dev, TURNON))
134                 tp->t_state |= CARR_ON;
135           else
136                 tp->t_state &= ~CARR_ON;
137           if (!(flag&FNDELAY))
138                 while ((tp->t_state&CARR_ON)= =0) {
139                       tp->t_state |= WOPEN;
140                       sleep((caddr_t)&tp->t_canq, TTIPRI);
141                 }
142           (*linesw[tp->t_line].l_open)(tp);
143           splx(x);
144     }
```

The code is defined as follows:

Line no.	Definition
112-114:	If the minor number indicates a device that does not exist, indicate the error and return.
118-120:	If the line is already open for exclusive use, and the current user is not the superuser, indicate the error and return.
122-128:	If the line is not already open, initialize the tty structure by means of a call to **ttinit**, 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.
130:	Defer interrupts so the interrupt routines cannot change the state while it is being examined.
131-136:	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.

137-140: If **open** is supposed to wait for the carrier, wait until the carrier is present.

142: Call the **l_open** routine indirectly through the *linesw* table. This completes the work required for the current line discipline to open a line.

143: Allow further interrupts.

tdclose: Lines 145 to 157

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

```
145
146   tdclose(dev)
147   {
148         register struct tty *tp;
149
150         tp = &td_tty[UNMODEM(dev)];
151         (*linesw[tp->t_line].l_close)(tp);
152         if (tp->t_cflag & HUPCL)
153               tdmodem(dev, TURNOFF);
154         tp->t_lflag &= ~XCLUDE;  /* turn off exclusive use bit */
155         /* turn off interrupts */
156         out(td_addr[UNMODEM(dev)] + RIENABL, 0);
157   }
```

The code is defined as follows:

Line no.	Definition
151:	Call the **close** routine through the *linesw* table to do the work required by the current line discipline.
152-153:	If the "hang up on last close" bit is set, drop the data-terminal-ready and request-to-send signals.
154:	Reset the exclusive-use bit.
156:	To prevent spurious interrupts, disable all interrupts for this line.

tdread and tdwrite: Lines 158 to 168

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.

```
158
159   tdread(dev)
160   {
161         (*linesw[tp->t_line].l_read)(&td_tty[UNMODEM(dev)]);
162   }
163
164   tdwrite(dev)
165   {
166
167         (*linesw[tp->t_line].l_write)(&td_tty[UNMODEM(dev)]);
168   }
```

tdparam: Lines 169 to 205

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

```
169
170   tdparam(dev)
171   {
172         register int cflag;
173         register int addr;
174         register int temp, speed, x;
175
176         addr = td_addr[UNMODEM(dev)];
177         cflag = td_tty[UNMODEM(dev)].t_cflag;
178
179         /* if speed is B0, turn line off */
180         if ( (cflag & CBAUD) = = B0){
181               outb(addr + RCNTRL, inb(addr+RCNTRL) & ~CDTR & ~CRTS);
182               return;
183         }
184
185         /* set up speed */
186         outb( addr + RSPEED, td_speeds[ cflag & CBAUD ]);
187
188         /* set up line control */
189         temp = (cflag & CSIZE) >> 4; /* length */
190         if ( cflag & CSTOPB )
191               temp |= CSTOP2;
192         if ( cflag & PARENB ) {
193               temp |= CPARITY;
194               if ( (cflag & PARODD) = = 0)
195                     temp |= CEVEN;
196         }
```

```
197         temp |= CDTR | CRTS;
198         out( addr + RCNTRL, temp );
199
200         /* setup interrupts */
201         temp = EXMIT;
202         if ( cflag & CREAD )
203               temp |= ERECV;
204         outb(addr + RENABL, inb(RENABL) | temp);
205   }
```

The code is defined as follows:

Line no.	Definition
176-177:	Get the base address and flags for the referenced line.
180-182:	The speed B0 means "hang up the line."
186-205:	The remainder of the **tdparam** routine simply loads the device registers with the correct values.

tdmodem: Lines 206 to 224

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.

```
206
207   tdmodem(dev, cmd)
208   int dev, cmd;
209   {
210         register int addr;
211
212         addr = td_addr[UNMODEM(dev)];
213         switch(cmd){
214         case TURNON: /* enable modem interrupts, set DTR & RTS true */
215               outb(addr + RENABL, inb(RENABL) | EMS);
216               outb(addr + RCNTRL, inb(RENABL) | CDTR | CRTS );
217               break;
218         case TURNOFF: /* disable modem interrupts, reset DTR, RTS */
219               outb(addr + RENABL, inb(RENABL) & ~EMS);
220               outb(addr + RCNTRL, inb(RENABL) & ~(CDTR | CRTS) );
221               break;
222         }
223         return (inb(addr + RSTATUS) & SDSR);
224   }
```

The code is defined as follows:

Line no.	Definition
214-217:	If *cmd* was TURNON, turn on modem interrupts, and assert data-terminal-ready and request-to-send.
218-221:	If *cmd* was TURNOFF, disable modem interrupts, then drop data-terminal-ready and request-to-send.
223:	Return a zero value if there is no data-set-ready on this line; otherwise return a nonzero value.

tdintr: Lines 225 to 251

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

```
225   #endif
226
227   tdintr(vec)
228   int vec;
229   {
230         register int iir, dev, inter;
231
232         switch( vec ) {
233               case VECT0:
234                     dev = 0;
235                     break;
236               case VECT1:
237                     dev = 1;
238                     break;
239               default:
240                     printf("tdint: wrong level interrupt (%x)\n",vec)
241                     return;
242               }
243         while( (iir = inb(td_addr[dev]+RIIR)) != 0) {
244               if( (iir & IXMIT) != 0 )
245                     tdxint(dev);
246               if( (iir & IRECV) != 0 )
247                     tdrint(dev);
248               if( (iir & IMS) != 0 )
249                     tdmint(dev);
250         }
251   }
```

The code is defined as follows:

Line no.	Definition
232-242:	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**.
243-251:	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 252 to 272

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.

```
252
253   tdxint(dev)
254   {
255         register struct tty *tp;
256         register int addr;
257
258         tp = &td_tty[UNMODEM(dev)];
259         addr = td_addr[UNMODEM(dev)];
260         if ( inb(addr + RSTATUS) & STRDY )
261         {
262               tp->t_state &= ~BUSY;
263               if (tp->t_state & TTXON) {
264                     outb(addr + RTDATA, CSTART);
265                     tp->t_state &= ~TTXON;
266               } else if (tp->t_state & TTXOFF) {
267                     outb(addr + RTDATA, CSTOP);
268                     tp->t_state &= ~TTXOFF;
269               } else
270                     tdproc(tp, T_OUTPUT);
271         }
272   }
```

The code is defined as follows:

Line no.	Definition
260:	If the transmitter is ready, reset the busy indicator.
263-265:	If the line is to be restarted, send a CSTART, and reset the indicator.
266-268:	If the line is to be stopped, send a CSTOP, and reset the character.
269-270:	Otherwise, call **tdproc** and ask it to send the next character in the queue.

tdrint: Lines 273 to 338

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.

```
273
274   tdrint(dev)
275   {
276         register int c, status;
277         register int addr;
278         register struct tty *tp;
279
280         tp = &td_tty[UNMODEM(dev)];
281         addr = td_addr[UNMODEM(dev)];
282
283         /* get char and status */
284         c = inb( addr + RRDATA );
285         status = inb(addr + RLSR);
286
287         /*
288          *  Were there any errors on input?
289          */
290         if( status & SOERR )            /* overrun error */
291               c |= OVERRUN;
292         if( status & SPERR )            /* parity error */
293               c |= PERROR;
294         if( status & SFERR )            /* framing error */
295               c |= FRERROR;
296
297         if (tp->t_rbuf.c_ptr == NULL)
298               return;
299         flg = tp->t_iflag;
300         if (flg&IXON) {
```

```
301                 register int ctmp;
302                 ctmp = c & 0177;
303                 if( tp->t_state & TTSTOP ) {
304                       if (ctmp == CSTART || flg&IXANY)
305                             (*tp->t_proc)(tp, T_RESUME);
306                 } else {
307                       if (ctmp == CSTOP)
308                             (*tp->t_proc)(tp, T_SUSPEND);
309                 }
310                 if (ctmp == CSTART || ctmp == CSTOP)
311                       return;
312           }
313           if (c&PERROR && !(flg&INPCK))
314                 c &= ~PERROR;
315           if (c&(FRERROR|PERROR|OVERRUN)) {
316                 if ((c&0377) == 0) {
317                       if (flg&IGNBRK)
318                             return;
319                       if (flg&BRKINT) {
320                             (*linesw[tp->t_line].l_input)
321                                         (tp, L_BREAK);
322                             return;
323                       }
324                 } else {
325                       if (flg&IGNPAR)
326                             return;
327                 }
328           } else {
329                 if (flg&ISTRIP)
330                       c &= 0177;
331                 else {
332                       c &= 0377;
333                 }
334           }
335           *tp->t_rbuf.c_ptr = c;
336           tp->t_rbuf.c_count--;
337           (*linesw[tp->t_line].l_input)(tp, L_BUF);
338     }
```

The code is defined as follows:

Line no.	Definition
283-285:	Get the character and status.
290-295:	If any errors were detected, set the appropriate bit in *c*.
300-312:	This code determines whether the character is X-ON and, if output is stopped, it restarts it. If the character is X-OFF, output is suspended.

313-334: Further error checking is then carried out and characters in error are discarded. The character is then placed in the queue.

335-336: This code stores the character into the received buffer and decrements the character count.

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

tdmint: Lines 339 to 366

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

```
339
340   tdmint(dev)
341   {
342         register struct tty *tp;
343         register int addr,c;
344
345         tp = &td_tty[UNMODEM(dev)];
346         if ( tp->t_cflag & CLOCAL ) {
347               return;
348         }
349         addr = td_addr[UNMODEM(dev)];
350
351         if (inb(addr + RSTATUS) & SDSR) {
352               if ((tp->t_state & CARR_ON)= =0) {
353                     tp->t_state |= CARR_ON;
354                     wakeup(&tp->t_canq);
355               }
356         } else {
357               if (tp->t_state & CARR_ON) {
358                     if (tp->t_state & ISOPEN) {
359                           signal(tp->t_pgrp, SIGHUP);
360                           tdmodem(dev, TURNOFF);
361                           ttyflush(tp, (FREAD|FWRITE));
362                     }
363                     tp->t_state &= ~CARR_ON;
364               }
365         }
366   }
```

The code is defined as follows:

Line no.	Definition
346-347:	If there is no modem support for this line, just return.

351-354: 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.

356-365: 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**, and mark the line as having no carrier.

tdioctl: Lines 367 to 376

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

```
367
368   tdioctl(dev, cmd, arg, mode)
369   int dev;
370   int cmd;
371   faddr_t arg;
372   int mode;
373   {
374         if (ttiocom(&td_tty[UNMODEM(dev)], cmd, arg, mode))
375               tdparam(dev);
376   }
```

tdproc: Lines 377 to 454

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.

```
377
378   tdproc(tp, cmd)
379   register struct tty *tp;
380   {
381         register c;
382         register int addr;
383
384         extern ttrstrt;
385
386         addr = td_addr[tp - td_tty];
387         switch (cmd) {
388
389         case T_TIME:
```

```
390                 tp->t_state &= ~TIMEOUT;
391                 outb(addr + RCNTRL, inb(addr + RCNTRL) & ~CBREAK);
392                 goto start;
393
394           case T_WFLUSH:
395                 tp->t_tbuf.c_size -= tp->t_tbuf.c_count;
396                 tp->t_tbuf.c_count = 0;
397           case T_RESUME:
398                 tp->t_state &= ~TTSTOP;
399                 goto start;
400
401           case T_OUTPUT:
402           start:
403                 if (tp->t_state&(TIMEOUT|TTSTOP|BUSY))
404                       break;
405                 {
406                       register struct ccblock *tbuf;
407
408                       tbuf = &tp->t_tbuf;
409                       if ( tbuf->c_ptr == NULL ||
410                             tbuf->c_count == 0 ) {
411                             if ( tbuf->c_ptr )
412                                   tbuf->c_ptr  -= tbuf->c_size
413                                                 - tbuf->c_count;
414                             if ( ! (CPRES &
415                                 (*linesw[tp->t_line].l_output)(tp)))
416                                         break;
417                       }
418                       tp->t_state |= BUSY;
419                       outb(addr + RTHR, *tbuf->c_ptr++);
420                       tbuf->c_count--;
421                 }
422                 break;
423
424           case T_SUSPEND:
425                 tp->t_state |= TTSTOP;
426                 break;
427
428           case T_BLOCK:
429                 tp->t_state &= ~TTXON;
430                 tp->t_state |= TBLOCK;
431                 if (tp->t_state&BUSY)
432                       tp->t_state |= TTXOFF;
433                 else
434                       outb(addr + RTDATA, CSTOP);
435                 break;
436
437           case T_RFLUSH:
438                 if (!(tp->t_state&TBLOCK))
439                       break;
440           case T_UNBLOCK:
441                 tp->t_state &= ~(TTXOFF|TBLOCK);
442                 if (tp->t_state&BUSY)
443                       tp->t_state |= TTXON;
```

```
444                 else
445                       outb(addr + RTDATA, CSTART);
446                 break;
447
448           case T_BREAK:
449                 outb( addr + RCNTRL, inb( addr + RCNTRL ) | CBREAK );
450                 tp->t_state |= TIMEOUT;
451                 timeout(ttrstrt, tp, Hz/4);
452                 break;
453           }
454     }
```

The code is defined as follows:

Line no.	Definition
387:	The *cmd* argument determines the action taken.
389-392:	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.
398-399:	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 402).
403-404:	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.
405-420:	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** 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.

424-426: 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.

428-435: 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.

437-439: 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.

Line no. **Definition**

440-446: 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.

448-452: 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.

10.4 Disk Drive Code Examples

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.

```
 1    /*
 2    ** hd- prototype hard disk driver
 3    */
 4
 5    #include "../h/param.h"
 6    #include "../h/buf.h"
 7    #include "../h/iobuf.h"
 8    #include "../h/dir.h"
 9    #include "../h/conf.h"
10    #include "../h/user.h"
11
12
13    /* disk parameters */
14    #define NHD  4      /* number of drives */
14a   #define NPARTS      8      /* # partitions/disk */
15    #define NCPD 600    /* # cylinders/disk */
16    #define NTPC 4      /* # tracks/cylinder */
17    #define NSPT 10     /* # sectors/track */
```

```
18  #define NBPS 512    /* # bytes/sector */
19  #define NSPB (BSIZE/NBPS)       /* sectors/block */
20  #define NBPC (NTPC*NSPT/NSPB)   /* blocks/cylinder */
21
22  /* addresses of controller registers */
23  #define RBASE       0x00        /* base of all registers */
24  #define RCMD (RBASE+0)    /* command register */
25  #define RSTAT (RBASE+1)   /* status - nonzero means error */
26  #define RCYL (RBASE+2)    /* target cylinder */
27  #define RTRK (RBASE+3)    /* target track */
28  #define RSEC (RBASE+4)    /* target sector */
29  #define RADDRL (RBASE+5) /* target memory address lo 16 bits*/
30  #define RADDRH (RBASE+6) /* target memory address hi 8 bits*/
31  #define RCNT (RBASE+7)    /* number of sectors to xfer */
32
33  /* bits in RCMD register */
34  #define CREAD       0x01        /* start a read */
35  #define CWRITE      0x02        /* start a write */
36  #define CRESET      0x03        /* reset the controller */
37
38  /*
39  ** minor number layout is 0000dppp
40  **  where d is the drive number and ppp is the partition
41  */
42  #define drive(d)      ( minor(d) >> 3)
43  #define part(d)       ( minor(d) & 0x07)
44
45  /* partition table */
46  struct partab {
47      daddr_t len;   /* # of blocks in partition */
48      int     cyloff;       /* starting cylinder of partition */
49  };
50
```

The code is defined as follows:

Line no.	Definition
14:	NHD defines the number of drives to which the controller can be attached.
14a:	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,     /* usr 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 *usr* areas.
64:	The buffer headers representing requests for this driver are linked into a queue, with *hdtab* forming the head of the queue. 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 102 to 131

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 strategy routine is responsible for validating the request, and linking it into the queue of outstanding requests.

```
102   int hdstrategy(bp)
103   register struct buf *bp;
104   {
105       register int dr, pa;     /* drive and partition numbers */
106       daddr_t sz, bn;
107       int x;
108       dr = drive(bp->b_dev);
109       pa = part(bp->b_dev);
110       bn = bp->b_blkno * NSPB;
111       sz = (bp -> b_bcount + BMASK) >> BSHIFT;
112       if ( dr<NHD && pa<NPARTS && bn>=0 && bn<hd_sizes[pa].len &&
113           ((bn + sz < hd_sizes[pa].len) || (bp->b_flags & B_READ)))
114       {
115           if ( bn + sz > hd_sizes[pa].len ) {
116               sz = (hd_sizes[pa].len - bn) * NBPS;
117               bp->b_resid = bp->b_bcount - (unsigned) sz;
118               bp->b_bcount = (unsigned) sz;
119           }
120       } else {
121           bp->b_flags |= B_ERROR;
122           iodone(bp);
123           return;
124       }
```

```
125         bp->b_cylin = (b_blkno / NBPC) + hd_sizes[pa].cyloff;
126         x = splbuf();
127         disksort(&hdtab, bp)
128         if (bp->b_active = = NULL)
129             hdstart();
130         splx(x);
131     }
```

The code is defined as follows:

Line no.	Definition
108-111:	First, compute various useful numbers that will be used repeatedly during the validation process.
112-124:	If the request is for a nonexistent drive or a nonexistent partition, if it lies completely outside the specified partition, 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** 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.
125:	Compute the target cylinder of the request for the benefit of the **disksort** routine.
126:	Block interrupts to prevent the interrupt routine from changing the queue of outstanding requests.
127:	Sort the request into the queue by passing it and the head of the queue to the **disksort** routine.
128:	If the controller is not already active, start it up.
129:	Re-enable interrupts and return to the user process.

hdstart: Lines 132 to 166

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

```
132
133     /*
134      *      Startup Routine:
```

```
135     *       Arguments:
136     *         None
137     *       Function:
138     *         Compute device-dependent parameters
139     *         Start up device
140     *         Indicate request to I/O monitor routines
141     */
142   hdstart()
143   {
144       register struct buf *bp;       /* BUFFER POINTER */
145       register unsigned sec;
146
147       if ((bp = hdtab.b_actf) = = NULL) {
148           hdtab.b_active = 0;
149           return;
150       }
151       hdtab.b_active = 1;
152
153       sec = ((unsigned)bp->blkno * NSPB);
154       out(RCYL, sec / NSPC);              /* cylinder */
155       sec %= NSPC;
156       out(RTRK, sec / NSPT);              /* track */
157       out(RSEC, sec % NSPT);              /* sector */
158       out(RCNT, bp->b_count / NBPS);      /* count */
159       out(RDRV, drive(bp->b_dev));        /* drive */
160       out(RADDRL, bp->b_paddr & 0xffff); /* memory address lo */
161       out(RADDRH, bp->b_paddr >> 16);    /* memory address hi */
162       if ( bp->b_flags & B_READ )
163           out(RCMD, CREAD);
164       else
165           out(RCMD, CWRITE);
166   }
```

The code is defined as follows:

Line no.	Definition
147-149:	If there are no active requests, mark the state of the driver as idle, and return.
151:	Mark the state of the driver as active.
153-157:	Calculate the starting cylinder, track, and sector of the request, and load the controller registers with these values.
159-161:	Load the controller with the drive number, and the memory address of the data to be transferred.
162-166:	If the request is a read request, issue a read command; otherwise, issue a write command.

hdintr: Lines 167 to 201

The **hdintr** routine is called by the kernel through the *vecintsw* table whenever the controller issues an interrupt.

```
167
168   /*
169    *      Interrupt routine:
170    *        Check completion status
171    *        Indicate completion to i/o monitor routines
172    *        Log errors
173    *        Restart (on error) or start next request
174    */
175   hdintr()
176   {
177       register struct buf *bp;
178
179       if (hdtab.b_active = = 0)
180           return;
181
182       bp = hdtab.b_actf;
183
184       if ( in(RSTAT) != 0 )
185           out(RCMD, CRESET);
186           if (++hdtab.b_errcnt <= ERRLIM) {
187               hdstart;
188               return;
189           }
190           bp->b_flags |= B_ERROR;
191           deverr(&hdtab, bp, in(RSTAT), 0);
192       }
193       /*
194        *      Flag current request complete, start next one
195        */
196       hdtab.b_errcnt = 0;
197       hdtab.b_actf = bp->av_forw;
198       bp->b_resid = 0;
199       iodone(bp);
200       hdstart();
201   }
```

The code is defined as follows:

Line no.	Definition
179-180:	If an unexpected call occurs, just return.
182:	Get a pointer to the first buffer header in the chain; this is the request that is currently being serviced.

184-192: 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 **deverror** to print a console message about the failure.

196-201: Mark this request complete, take it out of the request queue, and call **hdstart** to start on the next request.

hdread: Lines 202 to 222

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 strategy 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-strategy routine.

```
202
203   /*
204    *  raw read routine:
205    *    This routine calls "physio" which computes and validates
206    *    a physical address from the current logical address.
207    *
208    *      Arguments
209    *        Full device number
210    *      Functions:
211    *        Call physio which does the actual raw (physical) I/O
212    *        The arguments to physio are:
213    *          pointer to the strategy routine
214    *          buffer for raw I/O
215    *          device
216    *          read/write flag
217    */
218   hdread(dev)
219   {
220
221       physio(hdstrategy, &rhdbuf, dev, B_READ);
222   }
```

hdwrite: Lines 231 to 235

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.

```
223
224   /*
```

```
225     *       Raw write routine:
226     *       Arguments(to hdwrite):
227     *         Full device number
228     *       Functions:
229     *         Call physio which does actual raw (physical) I/O
230     */
231    hdwrite(dev)
232    {
233
234        physio(hdstrategy, &rhdbuf, dev, B_WRITE);
235    }
```

Appendix A

The Select System Call

A.1 Supporting the Select System Call

This section describes routines that support the **select**(S) system call. The routines are called:

selsuccess(), selfailure(), selread(), selwrite(), selexcept(), selwakeup()

Drivers that are to support the **select**(S) call must include global declarations like the following:

```
#include "../h/select.h"
extern int selwait;
struct xx_selstr
{
            struct proc *read;
            struct proc *write;
            struct proc *except;
            char flags;
}
xxselstr[NUM_MINOR_DEVS];
```

Syntax: selsuccess();

Description: A driver uses **selsuccess** to indicate that the condition which the user has selected is true, and the process should not block.

The **select**(S) system call code uses a unique ioctl 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 that is being selected, and has three valid values: SELREAD, SELWRITE and SELEXCEPT. These are defined in *select.h*. So all drivers that are to support **select** must implement the IOC_SELECT ioctl with code analogous to that displayed in the example section for **selwakeup**().

Syntax: selfailure();

Description: A driver uses **selfailure** to indicate that the condition which the user has selected is not true, and the process should block.

Syntax: selwakeup(procp, flags);
struct proc * procp;
char flags;

Description: 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: struct proc * procp;
char flags;

procp is a pointer to a process table entry which is found in the xxselstr data structure. Every 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** 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 **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;
      }

/*
* The condition is unsatisfied so the 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;
        }
}
```

Appendix B

Sharing Interrupt Vectors

B.1 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. The configuration utility, **config**, lets the user specify two devices to share an interrupt level. For more information about this procedure, see *config*(ADM) and *master*(F) in the *XENIX Reference Manual*.

If there are two devices, *aa* and *bb* that share interrupt level 3, the code in the *c.c* file generated by **config** should be as follows:

```
vector3(level)
int level;
{
      aaintr(level);
      bbintr(level);
}


int (*vecintsw[])() =
{
      clock,
      consintr,
      novec,
      vector3,
      novec,
      etc
      .
      .
      .
}
```

The interrupt routines **aaintr** and **bbintr** should have the following format:

```
xxintr(level)
int level;
{
      IF NOT MY INTERRUPT
            return;

      NORMAL INTERRUPT PROCESSING
}
```

Appendix C
Warnings

C.1 Warnings

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

- Do not defer interrupts with *spl5*( ) or other *spl* calls any longer than necessary.
- Do not change the per process data in the *u* structure at interrupt time.
- Do not call *seterror*( ) or *sleep*( ) at interrupt time.
- Do not set your priority level at interrupt time to a lower priority than the one at which your interrupt routine was called.
- Make interrupt time processing 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.
- 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.

Replace this Page
with Tab Marked:

Index

Index

F

G

H

I

10-31-88

SCO-514-210-051