XENIX* 286
DEVICE DRIVER GUIDE

*XENIX is a trademark of Microsoft Corporation.

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 174393-001

XENIX* 286
DEVICE DRIVER GUIDE

Order Number: 174393-001

*XENIX is a trademark of Microsoft Corporation.

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

REV.	REVISION HISTORY	DATE
-001	Original issue	11/84

intel®

TABLE OF CONTENTS

CONTENTS

CHAPTER 1	PAGE
INTRODUCTION	
Prerequisites	1-1
Manual Organization	1-1
Notation	1-2
CHAPTER 2	
DRIVER FUNDAMENTALS	
XENIX I/O Overview	2-1
Basic I/O Model	2-1
I/O Levels	2-2
Device Types	2-3
Driver Overview	2-3
Kernel Review	2-4
What Is the Kernel?	2-4
Privilege Levels	2-4
Memory Organization	2-4
Multiple Processes	2-5
Process Control	2-6
Interrupt Handling	2-7
Locking Out Interrupts	2-8
Device Identification	2-9
Device Driver Interface	2-11
Driver Files	2-12
Driver Support Routines	2-13
Physical I/O Routines	2-13
Accessing User Memory	2-14
CHAPTER 3	
SIMPLE CHARACTER DRIVERS	
Character Buffering	3-1
Driver Files	3-3
Driver Constants	3-3
Data Structures	3-4
Driver Procedures	3-5
ixxxinit Procedure	3-5
ixxxopen Procedure	3-5
ixxxclose Procedure	3-6
ixxxread Procedure	3-6
ixxxwrite Procedure	3-7
ixxxioctl Procedure	3-8
ixxxenq Procedure	3-8
ixxxstart Procedure	3-8
ixxxintr Procedure	3-9
Output Summary	3-10

CONTENTS **PAGE**

CHAPTER 4
TERMINAL DRIVERS
tty Structure 4-2
Line Discipline Routines 4-2
ttinit 4-3
ttopen 4-4
ttclose 4-4
ttread 4-4
ttwrite 4-4
ttiocom 4-5
ttioctl 4-5
ttin 4-5
ttout 4-6
ttxput 4-6
Modem Control by Terminal Drivers 4-6
Driver Description 4-6
ixxxinit Procedure 4-7
ixxxparam Procedure 4-7
ixxxopen Procedure 4-8
ixxxclose Procedure 4-8
ixxxread Procedure 4-9
ixxxwrite Procedure 4-9
ixxxintr Procedure 4-9
ixxxproc Procedure 4-10
ixxxstart Procedure 4-11
ixxxioctl Procedure 4-11
iSBC 534 Driver 4-12
sys/h/i534.h Listing 4-12
sys/cfg/c534.c Listing 4-17
sys/io/i534.c Listing 4-18

CHAPTER 5
BLOCK DRIVERS
Block Buffering 5-2
Block Driver Overview 5-6
Driver Files 5-8
Driver Constants 5-8
Driver Data Structures 5-10
ixxxinit Procedure 5-11
ixxxopen Procedure 5-12
ixxxclose Procedure 5-13
ixxxstrategy Procedure 5-14
ixxxstart Procedure 5-15
ixxxintr Procedure 5-16
ixxxread and ixxxwrite Procedures 5-17
ixxxioctl Procedure 5-18

CONTENTS PAGE

CHAPTER 6
ADDING DRIVERS TO THE CONFIGURATION
Editing the master file 6-2
Editing xenixconf 6-6
Editing the makefiles 6-7
Making a New Kernel 6-7
Making the Device Special File 6-8
Adding Terminal Information 6-8
Executing the New Kernel 6-9
Deleting a Device Driver 6-9

APPENDIX A
MEMORY-MAPPED I/O FOR DRIVERS
Small Model Kernel A-1
Creating the Segment Descriptor A-1
The Peek Routines A-2
The Poke Routines A-3

APPENDIX B
CONVERTING DRIVERS FROM
RELEASE 1 TO RELEASE 3 OF XENIX 286
Terminal Drivers B-1
tty Structure B-1
Changes to Routines B-4
Line Discipline Routines B-6
The tty.h File B-6
Block Device Drivers B-9
Buffer Changes B-9
Addressing B-9

APPENDIX C
tty.h INCLUDE FILE

APPENDIX D
termio.h INCLUDE FILE

APPENDIX E
buf.h INCLUDE FILE

APPENDIX F
iobuf.h INCLUDE FILE

APPENDIX G
master FILE

APPENDIX H
xenixconf FILE

APPENDIX I
c.c FILE

APPENDIX J
RELATED PUBLICATIONS

INDEX

TABLES

TABLE	TITLE	PAGE
2-1	XENIX I/O Software	2-2
B-1	Changed tty Fields	B-2
B-2	New tty Fields	B-3
B-3	ixxxproc Commands	B-4
B-4	Line Discipline Routines	B-6
B-5	Input Modes Describing Basic Terminal Input Control	B-7
B-6	Output Modes Specifying System Treatment of Output	B-7
B-7	Control Modes Describing Hardware Control of the Terminal	B-8
B-8	Line Discipline Modes Used to Control Terminal Function	B-8

FIGURES

FIGURE	TITLE	PAGE
6-1	Device Table from sys/conf/master	6-2

intel®

CHAPTER 1
INTRODUCTION

This manual describes how to write or modify device drivers for Intel's Release 3 of the XENIX 286 Operating System. A *device driver* is a software module that controls an input/output (I/O) device and provides a system-defined interface to the device.

Prerequisites

This manual assumes that you, the reader, understand the C programming language and basic programming concepts. This manual also assumes some knowledge of XENIX or UNIX. If you are writing or modifying a specific driver, you should also understand any hardware controlled by the driver.

Manual Organization

This manual contains these chapters:

1. **Introduction:** Prerequisites, manual organization, and notation.

2. **Driver Fundamentals:** XENIX I/O overview, device types, driver interface, driver organization, kernel concepts, and kernel support routines.

3. **Simple Character Drivers:** Character buffering services and how to write a simple character driver.

4. **Terminal Drivers:** Terminal support services and how to write a terminal driver.

5. **Block Drivers:** Block buffering, caching, and sorting services and how to write a block driver.

6. **Adding Drivers to the Configuration:** How to modify the XENIX 286 system to include your driver.

This manual contains these appendixes:

A. **Memory-Mapped I/O for Drivers:** Procedures used for reading and writing device registers that are mapped into the iAPX 286 main memory address space, rather than the iAPX 286 I/O port address space.

B. **Converting Drivers from Release 1 to Release 3 of XENIX 286:** Hints and guidelines for converting drivers written for Release 1 of Intel's XENIX 286 Operating System to Release 3.

C. **tty.h Include File:** definitions used by terminal drivers and other character drivers.

D. **termio.h Include File:** additional definitions used by terminal drivers.

E. **buf.h Include File:** definition of block buffer headers used for block I/O.

F. **iobuf.h Include File:** definition of static headers (one per block device) that reference lists of block buffer headers for block I/O.

G. **master File:** configuration file to be edited when defining a new device driver.

H. **xenixconf File:** configuration file to be edited when adding or removing devices in a configuration.

I. **c.c File:** C program file that contains the interfaces to all device drivers in the configuration.

J. **Related Publications:** Descriptions and ordering information for all XENIX 286 Release 3 manuals and any other publications referenced by this manual.

Notation

These notational conventions are used in this manual:

- Literal names are bolded where they occur in text, e.g., **/sys/include, printf, dev_tab, EOF.**
- Syntactic categories are italicized where they occur and indicate that you must substitute an instance of the category, e.g., *filename.*
- In examples of dialogue with the XENIX 286 system, characters entered by the user are printed in bold type, e.g., **cat myfile.**
- In syntax descriptions, optional items are enclosed in brackets, e.g., [**-n**].
- Items that can be repeated one or more times are followed by an ellipsis (**...**).
- Items that can be repeated zero or more times are enclosed in brackets and followed by an ellipsis (**[ ]...**).
- A choice between items is indicated by separating the items with vertical bars (|).
- Names of device driver routines and some data elements must use a device-specific prefix two to four characters in length. This prefix is represented in this manual as **ixxx**, the form of the prefix that Intel uses for its own devices. For example, the driver for Intel's iSBC® 544 board uses the prefix **i544.** Users writing drivers for Intel boards are encouraged to follow this convention.

intel®

CHAPTER 2
DRIVER FUNDAMENTALS

This chapter presents background information about XENIX I/O, the XENIX kernel, and XENIX device drivers that should be understood before reading subsequent chapters that cover the details of particular types of drivers. This chapter contains the following sections:

- XENIX I/O Overview (including device types and an overview of drivers)
- Kernel Review (including memory organization, process control, and interrupt handling)
- Device Identification
- Device Driver Interface
- Driver Files
- Driver Support Routines

XENIX I/O Overview

This section provides an overview of XENIX I/O:

- The basic model of I/O used by XENIX
- The four levels of software that handle I/O in XENIX
- Device types supported by XENIX
- An overview of device drivers

Basic I/O Model

Input/output in XENIX is part of a more general model of information transfer via *streams* of bytes. Such a stream may not even involve I/O devices, but may simply connect two user processes, one writing bytes to the stream and the other reading bytes from the stream. A stream can connect a process to a file, to a device, or via a "pipe" to another process. I/O is normally sequential, but for file I/O, it is possible to *seek* to a designated point in the file and then resume stream access.

(Note: A stream in this sense is more general than the "streams" defined by the XENIX standard I/O library, **stdio.**)

Most I/O sources and destinations are accessed via the XENIX hierarchical file system. Device interfaces are represented in the file system by *special files* that specify the device driver that implements I/O to or from the device. Special files are described later in this chapter in the section "Device Identification."

More information about the XENIX I/O model is contained in the *Overview of the XENIX 286 Operating System* and the *XENIX 286 C Library Guide.*

I/O Levels

Table 1-1 lists four types of software that provide I/O services in a XENIX system. Applications software can invoke three of the four types; device drivers are invoked only by the kernel and are never invoked directly by software outside of the kernel.

Table 2-1. XENIX I/O Software

Name	Description	I/O Services
Shell	Runs as a user process, provides a XENIX command language for use by users and by other programs; I/O services implemented with calls to standard I/O library.	Command syntax for I/O: redirection and *pipes* that connect processes using streams; also file system services such as *wildcards* in file names.
Standard I/O Library **stdio.h**	A library of C definitions used by C programs as a standard interface to I/O; implemented with calls to the XENIX kernel.	Open, read, write, close, seek files or devices; formatted I/O; *stream* I/O that interposes additional buffering* between program and file or device.
XENIX Kernel	The core of XENIX, the only code that users cannot replace as they choose; provides basic system services for memory management, timer management, process control, I/O, and system start/stop; calls device drivers for device-specific I/O functions.	Implements file system; block buffering* and caching; character buffering*; terminal line editing; vectoring of device interrupts.
Device Driver	Provides a XENIX-defined interface to a particular device.	Device-dependent code for initialization, startup, shutdown, read or write, interrupt handling, and device control.

*Buffering in the standard I/O library and buffering in the kernel are separate facilities.

Device Types

XENIX recognizes two types of device interfaces, *character* and *block*.

A character device reads or writes sequential streams of characters and is optimized for the transfer of a few characters with each operation. A line printer or a terminal are examples of character devices. Chapter 3 describes a simple character driver.

A block device is presumed to contain storage, organized as a randomly addressable array of blocks. Any block device can contain a XENIX file system. Disk drives, bubble memories, and RAM disks are all block devices.

Any device that does not fit the model of a block device is implemented as a character device, e.g., a driver for a local area network.

XENIX provides special support for terminal character devices. Much of the code needed in a terminal device driver is already provided by XENIX as a set of special *line discipline* routines. Chapter 4 describes terminal drivers.

The kernel provides extensive block buffering, caching, and sorting services to minimize and optimize I/O operations for block devices. For some operations, such as a byte-by-byte copy of a disk, this kernel support is inappropriate. To support such low-level operations, most block devices support a character interface to the device, sometimes called the "raw" interface, in addition to the block interface. Chapter 5 describes both interfaces to block devices.

Driver Overview

A XENIX device driver is organized as a set of procedures to be called by the kernel. The forms of the procedure names, the procedure parameters, the times the procedures are called, and the purposes of the procedures are all predetermined by the interface between the driver and the rest of the kernel. Many of the procedures are optional; some are required.

Within the driver procedures, the driver writer often relies on calls to kernel routines that provide needed services, such as process control, interrupt lockout, buffer manipulation, or I/O port operations. Some of these routines are described in subsequent sections of this chapter; others are described in Chapters 3, 4, and 5, which deal with specific types of drivers.

The driver writer must work with three constraints in writing the driver: the kernel interface to the driver, the kernel support routines available to the driver, and the characteristics of the hardware being controlled. A major purpose of this manual is to completely describe the first two design constraints; hardware-specific information is needed to describe the third design constraint.

Kernel Review

This section describes features of the XENIX 286 kernel that should be understood before writing device drivers. These features include memory organization, support for multiple processes, interrupt handling, and the kernel interface to device drivers.

What Is the Kernel?

The *kernel* is a small but central part of the total XENIX 286 Operating System. The purpose of the kernel is to provide a standard interface between other programs and shared machine resources. These resources include memory, processing time, and I/O devices. User programs and other parts of XENIX access the kernel through a set of standard *system calls*, defined as C function calls. XENIX command programs, such as the shell or the text editors, are outside the kernel and call on the kernel in the same way as any other user program.

Driver procedures and data structures are actually linked into the kernel as part of adding drivers to the configuration. Driver code executes in kernel mode, in the kernel address space, and with kernel privileges. The term *kernel* is used in two senses: (1) the device-independent kernel code that is linked with all the drivers; (2) *all* kernel-mode code, including all the drivers and the device-independent code.

Privilege Levels

Programs executing on an iAPX 286 processor have an associated *privilege level*. The privilege level ranges from 0 (most privileged/most trusted) to 3 (least privileged/least trusted). All kernel code and device driver code executes at level 0 (most privileged), allowing access to all of memory and execution of any iAPX 286 instruction. All other code in a XENIX 286 system executes at level 3 (least privileged), which restricts memory access and instruction execution to protect user processes from each other and to protect the kernel from user processes. The system call mechanism makes the transition between user code and privileged kernel code. A process is in *user mode* when it executes outside the kernel; a process is in *kernel mode* when it executes within the kernel (typically when it makes a system call).

Because device driver code is privileged, it must be carefully written and checked. Bad device driver code can corrupt user code or data, crash the system, or subvert system protection mechanisms.

Memory Organization

Memory in an iAPX 286 system is organized as a collection of segments. A particular process or program module may only be able to access certain segments; the segments that a process or module can access are its *address space*. Each process executing in user mode has a distinct address space, distinct from all other user processes and also distinct from the kernel address space.

Processes executing in kernel mode share a single *kernel address space*. However, at most one process can execute kernel code at a particular time. The kernel address space is also used by all interrupt handlers. Kernel code can use privileged iAPX 286 instructions to access user process address spaces in addition to accessing the kernel address space.

Multiple Processes

XENIX supports concurrent execution of multiple processes. The kernel is responsible for scheduling and coordinating processes. A process yields the CPU to another process for one of two reasons, either because it must wait for some event, such as I/O, before continuing, or because its time slice expires.

All processes in a XENIX system are handled in the same way by the kernel. There is no difference between "system" and "user" processes. Whenever *any* process executes a system call, it executes kernel code in kernel mode but retains its separate process identity.

Each process in a XENIX system is represented in the kernel address space by two data structures. First is an entry in the kernel's **proc** table, which contains information about all processes in the system. The second structure is a **u** structure that contains information the kernel needs to maintain about the process. The **u** structure also contains the kernel stack segment for the process, used for local variables, saved registers, and return addresses. Because kernel stack segments have a limited size, driver routines should not declare arrays or structures as local variables. The file **sys/h/user.h** defines the **u** structure.

Inactive processes may be swapped out to disk and their associated memory freed. When a process is swapped out, its kernel stack segment and **u** structure are swapped out with it. When the kernel is ready to run the process, the process is swapped into memory again, possibly in a different location than it used previously. The **proc** table entry is not swapped out or in and exists as long as the process exists.

Some information in the **u** structure is used by device drivers. Driver routines such as **ixxxwrite**, called by a process executing kernel code, can access the **u** structure. The driver interrupt routine or routines that can be called at interrupt time cannot access the **u** structure; at the time the interrupt executes for a particular process, that process may be swapped out and another process and *its* **u** structure swapped in.

The time when interrupt handling code is being executed is called *interrupt time*. All other execution time in a XENIX system is called *task time*. All interrupt time code is kernel-mode code (kernel or driver code). Task time code can be user-mode code, kernel code, or driver code. Task time code is executed on behalf of the currently running process. Kernel-mode task time code can access the process's **u** structure. Interrupt time code uses the kernel stack of whatever process was interrupted, but may be executing on behalf of an entirely different process. Code that may be called at interrupt time should never reference the **u** structure or process memory.

Process Control

When a process is executing in the kernel, it will not be pre-empted by any other process. Interrupts may be handled, but control returns to the interrupted process, even if a higher priority process has been readied. Only when a process exits the kernel or calls **sleep** can it be pre-empted, either by a higher priority process or by the expiration of its time slice.

When a device driver routine detects a situation in which the executing process must wait for some event, the routine must suspend process execution by calling a kernel process control routine. For example, if a process is doing input and must wait for additional characters to be sent from a terminal, then the driver routine should suspend the process so that other processes can run while the first process is waiting for I/O. When the driver detects that the process should again be able to run (e.g., a line has been received from the terminal), then the driver must "wake up" the suspended process, which will resume execution at the point in the driver where it was suspended.

These kernel process control routines are used in writing device drivers:

```
sleep(id, pri)
char *id;     /* unique address that the process is sleeping on;
              normally the address of a data structure used only by
              the process or routine that is sleeping. */
int pri;      /* process priority that the process will have when it
              wakes up and until it exits the kernel, when its normal
              priority is restored. */

wakeup(id)
char *id;     /* unique address that processes sleep on */
/*
All processes sleeping on the specified address are awakened.
The scheduler will dispatch them one at a time to resume executing
in the kernel, with the priority specified in their call to sleep.
The interrupt priority for an awakened process is spl0 (all interrupts
enabled).
*/
```

The priority specified to **sleep** must be in the range 0-127. Numerically lower values in this range indicate higher priority processes. If the priority specified in calling **sleep** is greater than or equal to **PZERO,** then the sleeping process can also be awakened by signals. **PZERO** is defined in **sys/h/param.h.**

Because a process can normally be awakened by signals or be awakened with another process waiting on the same address, it is good practice to recheck the condition being waited for after the process resumes execution, e.g.:

```
while (/* true if need to sleep */)
        sleep(&my_var, MY_PRI);
```

The call to **wakeup** the sleeping process is often in the device interrupt routine, **ixxxintr,** which detects the I/O event being waited for.

Interrupt Handling

In interrupt-time execution, an interrupt is handled by the kernel, which calls an interrupt handler. All interrupt handling is done in kernel mode. The kernel provides the special code needed to save and restore registers of the interrupted process and also provides the code to handle the 8259A PICs (Programmable Interrupt Controllers). Each device interrupt has two attributes: *level* and *priority*. The iAPX 286, PICs, and MULTIBUS® system bus support up to 256 interrupt levels, from 0 to 255. The two PICs provided on the iSBC 286/10 processor board support up to 15 interrupt levels. Seven of these interrupt levels can be used by device controllers on other MULTIBUS boards. Additional interrupt levels can be used by providing slave PICs on the other MULTIBUS boards, which send a level value to the 286/10 master PIC after signaling an interrupt. The interrupt level (sometimes called "vector") identifies the source of an interrupt and is passed as a parameter to the appropriate device interrupt routine. The level may only identify the type of device and possibly the board that interrupted and the interrupt routine may have to poll devices of the type or on the board to determine which devices need servicing.

Interrupt configuration for the iSBC 286/10 single board computer is described in *Guide to Using the iSBC 286/10 Single Board Computer*. The following information is taken from that guide and does not apply to any other 286 processor board that you may be using. As shipped by Intel, the 286/10 interrupts are configured as follows:

Level	PIC Level	Source
0	MASTER 0	Clock
1	MASTER 1	INT1 from MULTIBUS system bus
2	MASTER 2	INT2 from MULTIBUS system bus
3	MASTER 3	INT3 from MULTIBUS system bus
4	MASTER 4	INT4 from MULTIBUS system bus
5	MASTER 5	INT5 from MULTIBUS system bus
6	MASTER 6	8274 serial controller (console) interrupt
7	MASTER 7	SLAVE PIC interrupt
8-63	--------	---------
64	SLAVE 0	INT6 from MULTIBUS system bus
65	SLAVE 1	INT7 from MULTIBUS system bus
66	SLAVE 2	Jumper E145 in 286/10 interrupt matrix
67	SLAVE 3	MINTR0 from iSBX bus connector J6
68	SLAVE 4	MINTR1 from iSBX bus connector J6
69	SLAVE 5	MINTR0 from iSBX bus connector J5
70	SLAVE 6	MINTR1 from iSBX bus connector J5
71	SLAVE 7	Interrupt signal from the line printer interface

Note the jump in the interrupt level seen by the software ("Level" column above) when going from the master to the slave PIC. Because potentially each master line could be connected to a slave PIC, the level used by a slave PIC input line i is equal to $i+8*(j+1)$, where j is the number of the master PIC line to which the slave PIC is connected. For example, if your device interrupts on MULTIBUS line INT7, then it will use interrupt level 65 and should specify that level in the **master** file, as described in Chapter 6 of this manual.

INT0 from the MULTIBUS system bus is not listed above; it is connected to the 80286 Nonmaskable Interrupt (NMI) input.

A transaction being handled by an interrupt routine may be on behalf of a totally different process than the running process; the running process may not be involved in I/O to or from the interrupting device at all. Thus the interrupt routine and routines that can be called by the interrupt routine should not reference the **u** structure or reference process memory, as such references are erroneous and can corrupt other processes in the system.

Routines that can be called at interrupt time should not suspend the current process by calling **sleep.** Again, this prohibition is because the current process may not be the process that is being served by the interrupt-time routines.

Locking Out Interrupts

XENIX uses an 8-level priority scheme for managing interrupts. The routines for managing interrupt priorities and mutual exclusion and how they are implemented on Intel's hardware are described in this section.

The kernel provides 11 routines for locking and unlocking interrupts: **spl0, spl1, spl2, spl3, spl4, spl5, spl6, spl7, splx, splcli,** and **splbuf.**

A routine **spl***n* (*n* in the range 1-7) takes no arguments and locks out all interrupts of priority *n* or lower, enabling interrupts of priority *n*+1 or higher. **spl***n* returns an **int** value that is a mask to be used in restoring the previous interrupt priority. The returned value should be saved in a local variable for use as a parameter in a matching call to **splx.** The routine **spl0** enables all interrrupts, including priority 0 interrupts.

The routine **splx** takes as its argument the **int** mask value returned by an **spl***n* call. **splx** restores the priority to which interrupts were locked out *before* the matching **spl***n* call. There is no return value from **splx.** The mask value used should not be tampered with by the calling routine.

The routine **splcli** is a synonym for the routine **spl5** and locks out interrupts for all character devices and lower priorities. Character drivers should use this routine to lock out interrupts. (Block devices that have character interfaces do *not* count as character devices here.)

The routine **splbuf** is a synonym for **spl6** and locks out interrupts for all block devices and lower priorities. Block drivers should use this routine to lock out interrupts.

When the kernel services an interrupt, it locks out all interrupts at the same *level* or higher levels. For example, an interrupt at level 6 locks out all interrupts from level 6 or higher. The **spl** priority scheme maps to levels as follows: **spl7** locks out all interrupts. **splbuf (spl6)** locks out all but level 0 (clock) interrupts. **splcli (spl5)** locks out all but level 0 and level 1 interrupts. You should not be using **spl4, spl3, spl2,** or **spl1,** and their action is not defined in this manual. **spl0** enables interrupts at all levels.

Note that only a block device should use interrupt level 1. Also note that the priority mechanism is implemented entirely with the PICs; the 80286 CPU distinguishes only between all interrupts enabled and all interrupts (except NMI and RESET) disabled.

It is a programming error for driver code to ever lower the interrupt priority below its value when the driver code is called. Driver code should only use **splcli** or **splbuf** to raise the interrupt priority and **splx** to restore the interrupt priority.

The only driver code that must lock out interrupts is code that requires exclusive access to a data structure accessed at both task time and interrupt time. For example:

```
int msk;
...
msk = splcli();      /* Lock out character interrupts and lower. */
/* Put code here to access shared data structure. */
splx(msk);           /* Restore the previous interrupt state. */
```

Note that **splx** must be called on *every* path out of the code in which the interrupt priority is raised. The following code illustrates a common mistake:

```
int msk;
...
msk = splcli();
...
if (/* error condition */) {
        u.u_error = EIO;
        return;

}
...
splx(msk);
```

Note that if the error return is taken, the previous interrupt priority has not been restored.

The duration that interrupts are locked out should always be minimized.

Device Identification

Each distinct device in a XENIX system is identified by a node in the XENIX file system called a *device special file*. By convention, all device special files are contained in the **/dev** directory of the file system. Because devices are implemented as files, all the XENIX file protection mechanisms and file naming mechanisms can be used with devices. The special file for a device has a name, specifies whether the device is a block or character device, and specifies a *major number* and a *minor number* for the device.

The major number is eight bits and specifies a particular device driver. For example, the system console and the line printer are both character devices but have different major numbers and different drivers. The minor number designates a specific device within the class of devices handled by a driver. For example, if a system supports four line printers, they would use the same driver and major number, but use four different minor numbers, such as 0, 1, 2, and 3. The minor number is not interpreted by the kernel, only by the driver. The driver can use the minor number to encode information. For example, terminal drivers use bit 6 of the minor number to indicate whether modem control is enabled for a line. Hard disk drivers typically reserve some bits of the minor number to specify disk partitions (described in Chapter 5, "Block Drivers"). In contrast, the major number is normally ignored by a driver, and a driver should be written to be independent of whatever major number it is assigned.

The kernel combines the major and minor numbers into a 16-bit *device number*. The high byte of the device number is the major number; the low byte of the device number is the minor number. The macros **major** and **minor**, defined in **sys/h/types.h**, are used to extract the major and minor number from a device number. Each macro takes a device number as its single argument. For example:

```
#include "../h/types.h"          /* code file must be in a sibling directory
                                    of sys/h */

bozo(dev)
dev_t dev;          /* device number, major/minor */
{
int maj = major(dev);
int min = minor(dev);
...
}
```

The device number type **dev_t** is also defined in **sys/h/types.h**. Also defined in the file is the macro **makedev(x,y)**, which returns the device number formed from the major number **x** and the minor number **y**.

A different special device file is specified for each major/minor combination. For example, a system with four line printers would have four corresponding special files in the **/dev** directory: **lp0**, **lp1**, **lp2**, and **lp3**. If a device supports both block and character interfaces, then different special files denote the two different interfaces, though the major and minor numbers are the same.

When a program opens a device and establishes an I/O channel to or from the device, it specifies the name of the device special file. The kernel then determines the device number to be used for all calls to the device driver and also whether to use the table of character device drivers or block device drivers in making driver calls.

Chapter 6, "Adding Drivers to the Configuration," describes how to create the special files needed by your driver.

Device Driver Interface

The kernel interface to device drivers is a set of tables containing the addresses of device-specific routines. These tables are called *device switches.* The tables are

- **dinitsw** Routines to be called at system initialization to initialize devices.
- **vecintsw** Device interrupt routines; the table is indexed by hardware interrupt level (from the PIC).
- **cdevsw** Device routines (e.g., **ixxxwrite**) called in response to system calls for character device interfaces. **cdevsw** is indexed by major device number.
- **bdevsw** Device routines (e.g., **ixxxstrategy**) called in response to system calls for block device interfaces. **bdevsw** is indexed by major device number.

Note that example driver routine names and data names in this manual are of the form **ixxx...** in accordance with Intel's naming of its own drivers (e.g., **i534** for the prefix of names associated with the iSBC 534 board).

The driver routines called through **dinitsw** have no parameters and have names of the form **ixxxinit.** The **dinitsw** routines are called at system initialization time. At system initialization, interrupts have not yet been enabled, so the **ixxxinit** routines must not rely on interrupts occurring (i.e., don't use **sleep, timeout,** or any driver interrupt routines).

When an interrupt occurs, **vecintsw** is indexed with the MULTIBUS interrupt level and the corresponding routine is called. These routines have the form

```
ixxxintr(level)
int level;      /* MULTIBUS interrupt level, in range 0-255 */
```

cdevsw and **bdevsw** are each indexed by major device number and contain routines called in response to various system calls. **cdevsw** is used for character device interfaces; **bdevsw** is used for block device interfaces. The routines addressed by **cdevsw** are

ixxxopen(dev, oflag);
: Called each time a device handled by the driver is opened. Does error checking to validate the open; should enable interrupts and assign a device status variable to indicate that the device is open.

ixxxclose(dev, oflag);
: Called for the *last* close of a device handled by the driver. Should wait for all pending I/O for the device to complete, disable interrupts, and assign a device status variable to indicate that the device is closed.

ixxxread(dev);
: Called to read a number of bytes from a device handled by the driver into an area in user memory. The **u** structure specifies the number of bytes and the transfer address.

ixxxwrite(dev);
Called to write a number of bytes from an area in user memory to a device handled by the driver. The **u** structure specifies the number of bytes and the transfer address.

ixxxioctl(dev, cmd, arg, mode);
Called to execute some special function of a device handled by the driver, such as formatting a disk or changing the baud rate of a terminal.

bdevsw includes the same **ixxxopen** and **ixxxclose** routines as **cdevsw**, and one other routine:

ixxxstrategy(bp);
Called with a pointer to a buffer with data to be read or written. Checks the request for validity, queues it, and starts the device if it is idle.

ixxxstrategy is used for both reading and writing blocks.

bdevsw also includes one data structure reference, to **ixxxtab**, the first static buffer header for the block device.

More information about all these routines and their parameters is in Chapters 3, 4, and 5 for character interfaces and Chapter 5 for block interfaces.

Driver Files

Code for a device driver is typically contained in three files:

- **sys/h/ixxx.h** is an include file that defines constants and sometimes structure declarations used by the driver.
- **sys/cfg/cxxx.c** is a C program file that defines data structures used by the driver, especially configuration-dependent data structures.
- **sys/io/ixxx.c** is a C program file that defines driver routines required by the kernel interface and any internal driver routines called by the required routines. This file references the data structures defined in **sys/cfg/cxxx.c** as externals.

The file **sys/h/ixxx.h** is normally included by the other two files:

```
#include "../h/ixxx.h"
```

Depending on the type of driver, **sys/io/cxxx.c** and **sys/io/ixxx.c** must include other **.h** files, as described in Chapters 3, 4, and 5, which describe different types of drivers.

The **sys** directory is normally contained in the root directory and has an absolute path name of **/sys**. However, if your system has the XENIX 286 source product, then the **sys** directory is contained in the **/usr** directory and has an absolute path name of **/usr/sys**.

To add a driver to your system, you must modify several other files, as described in Chapter 6, "Adding Drivers to the Configuration."

Driver Support Routines

This section describes several kernel routines frequently used by device drivers. Kernel routines used only by character device drivers are described in Chapters 3 and 4. Kernel routines used only by block device drivers are described in Chapter 5. Process control routines are described in the section "Process Control" earlier in this chapter. Interrupt control routines are described in the section "Locking Out Interrupts" earlier in this chapter.

Physical I/O Routines

The kernel provides four functions that allow drivers to directly address iAPX 286 I/O ports:

```
int in(port)
unsigned port;          /* port address, in range 0-65535 */
/*
Return a 16-bit word read from I/O addresses port and port + 1.
*/

char inb(port)
unsigned port;          /* port address, in range 0-65535 */
/*
Return the byte read from the I/O port.
*/

out(port, value)
unsigned port;          /* port address, in range 0-65535 */
int value;              /* 16-bit value to be written */
/*
Write the 16-bit value to I/O addresses port (low byte) and
port + 1 (high byte).
*/

outb(port, value)
unsigned port;          /* port address, in range 0-65535 */
char value;             /* byte to be written */
/*
Write the byte value to the I/O port.
*/
```

These routines use privileged iAPX 286 instructions and can be called only from kernel or device driver code. If reading or writing a nonexistent port, these routines still function and do not hang or fault. Writing a nonexistent port has no effect. Reading a nonexistent port returns an indeterminate value, typically all ones (0xff or 0xffff) or all zeros (0).

Some device drivers may use memory-mapped I/O, in which device registers are mapped into the main memory address space of the iAPX 286, instead of being assigned addresses in the I/O port address space of the iAPX 286. For example, this technique is used by Intel's 188/48 device driver, which is supplied with XENIX 286. Memory-mapped I/O is described in Appendix A, "Memory-Mapped I/O for Drivers." The example code in Chapters 3, 4, and 5 all assumes that I/O ports are used for device registers.

Accessing User Memory

The kernel provides the routines **copyin** and **copyout** to move blocks of data from or to user memory:

```
copyin(src, dst, cnt)
faddr_t     src;        /* far pointer into user data segment      */
caddr_t     dst;        /* near pointer into kernel data segment   */
unsigned    cnt;        /* nonzero number of bytes to transfer     */
/*
Copies cnt bytes from the user data area referenced by src to
the kernel data area referenced by dst.  cnt cannot be 0 (or 64K
bytes would be copied).
*/


copyout(src, dst, cnt)
caddr_t     src;        /* near pointer into kernel data segment   */
faddr_t     dst;        /* far pointer into user data segment      */
unsigned    cnt;        /* nonzero number of bytes to transfer     */
/*
Copies cnt bytes from the kernel data area referenced by src to the
user data area referenced by dst.  cnt cannot be 0 (or 64K bytes
would be copied).
```

Near pointers are used for references within the kernel because the kernel (including all driver code) is compiled using small model, which uses a single data segment. The type **faddr_t** is defined in **user.h** and contains a segment selector in the high word and an offset in the low word. The type **caddr_t** is defined in **types.h** and is an **unsigned short** 16-bit value, the offset into the kernel data segment.

In a process's **u** structure, **u.u_base** has the type **faddr_t**, references a buffer in user memory (during a **read** or **write** system call), and can be used with **copyin** or **copyout**. For example:

```
char *my_buffer;
. . .
/* Copy bytes to be written from user memory to kernel memory */
copyin(u.u_base, my_buffer, u.u_count);
```

The **u** structure is defined in the file **sys/h/user.h**.

intel®

CHAPTER 3
SIMPLE CHARACTER DRIVERS

This chapter describes the elements of a simple character device driver. This driver is for an output-only device such as a line printer. This chapter first describes character buffering services provided by the kernel and used by the driver. It then describes the files, constants, data structures, and procedures that make up the driver. Some example code is included in this chapter; this code is for a hypothetical device and not for any real device supported by XENIX.

Character Buffering

Character device drivers queue character streams for input or output using the **clist** data structure declared in the file **sys/h/tty.h:**

```
struct clist {
        int     c_cc;                   /* character count */
        struct cblock       *c_cf;      /* pointer to first */
        struct cblock       *c_cl;      /* pointer to last */
};
```

The queued characters are contained in one or more blocks of type **cblock** (also declared in **tty.h**). The **clist** structure references a linked list of these blocks that contain the characters in the queue. The **clist** structure references the first and last blocks in the list and contains a count of the total number of characters queued in all the blocks.

Each block contains a link to the next block in the list (**NULL** if no next block), an area that can contain up to 24 queued characters, and indices to the first and last queued characters in that area.

Driver code need not access the **cblock** and **clist** structures, except for the field **c_cc** which gives the count of queued characters. The kernel routines **getc** and **putc** perform all needed operations on these structures. The **tty.h** file that declares these structures is listed in Appendix C of this manual.

The kernel maintains a single free list of blocks available for character buffering. This free list is shared by all character drivers. When a character is added to a queue and a new block is needed, one is obtained from the free list. When the last character is removed from a block, that block is returned to the free list. The **getc** and **putc** routines handle all the details of allocating and returning free blocks.

The number of characters in a single driver queue should be limited by a "high-water mark" to keep any single device from exhausting the space available for character buffering. A task trying to write to the device is suspended if the high-water mark is reached, to be awakened when the number of characters queued for output drops to a "low-water mark." The goal is to keep both the task and the driver active, while limiting use of character buffering space. The example driver in this chapter uses fixed marks declared in the driver. Terminal drivers typically use arrays of marks, **tthiwat** and **ttlowat** in **tty.h**, indexed by terminal baud rate.

The kernel routine **putc** adds a character to a queue:

```
int putc(c, q)
int c;
struct clist *q;
```

putc tries to add character **c** at the end of the queue referenced by **q**. **putc** returns 0 if the character is successfully enqueued. If there is no space in the queue and no more **cblocks** on the free list, the character is not enqueued and **putc** returns -1.

If -1 is returned, the recommended action is for the calling routine to call **sleep**, specifying **q** as the address to sleep on. (This is only recommended in routines that are never called at interrupt time; **sleep** should never be called at interrupt time.) When the interrupt routine drains the queue to the low-water mark, it can call **wakeup**, specifying **q** as the wakeup address.

The kernel routine **getc** removes a character from a queue:

```
int getc(q)
struct clist *q;
```

If the queue referenced by **q** is empty, **getc** returns -1; otherwise it removes and returns the next character in the queue.

If **my_q** is a queue header declared with

```
struct clist my_q;
```

then the number of characters in the queue can be referenced as **my_q.c_cc** and the queue can be initialized with this assignment:

```
my_q.c_cc = 0;
```

Note that it is not necessary to initialize the queue character pointers, which are not used when the queue is empty.

The **clist** data structure is accessed at both task-time and interrupt-time. The **getc** and **putc** routines handle all needed mutual exclusion, ensuring that character device interrupts are locked out when queues are manipulated.

Driver Files

The driver code is contained in three files:

- **sys/h/ixxx.h** defines constants and possibly structure declarations used by the driver.
- **sys/cfg/cxxx.c** defines data structures used by the driver, including configuration structures that may be modified as part of system configuration.
- **sys/io/ixxx.c** defines the driver routines.

The file **sys/h/ixxx.h** is included by the other two files:

```
#include "../h/ixxx.h"
```

The main driver file **/sys/io/ixxx.c** should also include these files:

```
#include    "../h/param.h"    /* for system parameters          */
#include    "../h/dir.h"      /* needed by other .h files used  */
#include    "../h/types.h"    /* for system data types          */
#include    "../h/tty.h"      /* for clist structure            */
#include    "../h/user.h"     /* u structure  and error codes   */
```

Adding the device to the configuration also requires editing the files **sys/conf/master** and **sys/conf/xenixconf,** as described in Chapter 6, "Adding Drivers to the Configuration."

Driver Constants

Driver constants typically defined in **sys/h/ixxx.h** include

1. The number of boards (**ixxx_BRD**) and the number of devices (**ixxx_NUM**) supported by the driver. The range of minor device numbers allowed is then 0 - (**ixxx_NUM** - 1).

   ```
   #define    ixxx_BRD   2      /* number of boards   */
   #define    ixxx_NUM   2*ixxx_BRD /* number of devices (2 per board) */
   ```

2. The interrupt level used by the driver, e.g., **ixxx_LEV.**

3. Offsets from the base port address for a particular device to the individual port addresses used by the device controller. For example:

   ```
   #define    ixxx_DAT   0      /* offset to data port     */
   #define    ixxx_STS   1      /* offset to status port   */
   ```

4. Constants that define possible state values for the device. For example:

```
#define   ixxx_ABS  0       /* device absent status   */
#define   ixxx_PRE  1       /* device present status  */
#define   ixxx_OPN  2       /* device open status     */
#define   ixxx_ERR  3       /* device error status; device must be closed
                               and reopened */
```

5. Constants that define bit patterns used in accessing the device (hardware dependent):

```
#define   ixxx_ONL  0001    /* online bit in status port     */
#define   ixxx_BSY  0002    /* device busy; cleared when
                               interrupt acknowledged  */
#define   ixxx_HER  0004    /* device hard error             */
#define   ixxx_INE  0100    /* device interrupts enabled     */
```

6. A constant that defines the task priority to be used when returning from a call to **sleep:**

```
#define   ixxx_PRI  TTOPRI  /* task priority used on return from sleep
                               (defined in tty.h)            */
```

7. High- and low-water marks for the device (if arrays declared in **tty.h** are not used):

```
#define   ixxx_LOW    48    /* low-water mark                */
#define   ixxx_HIW    96    /* high-water mark               */
```

Data Structures

Data structures typically defined in **sys/cfg/cxxx.c** include

1. An array of base port addresses for the devices, e.g.:

```
unsigned ixxx_adr[ixxx_NUM] = { 0140, 0150, 0160, 0170 };
```

The initialization clause can be edited to change the port addresses used. The number of port addresses listed must be greater than or equal to **ixxx_NUM.**

2. An array to record device status (e.g., absent, present, open, error):

```
int ixxx_sts[ixxx_NUM];  /* C initializes to zeros = all absent */
```

3. An array of character queue headers for the devices:

```
struct clist ixxx_q[ixxx_NUM];  /* ixxxinit must init count fields */
```

When configuring a system and assigning port addresses, a system administrator would edit the **sys/cfg/cxxx.c** file to define the port addresses assigned to **ixxx_adr.** The system administrator may also edit the **sys/h/ixxx.h** file to change the number of boards. Note that the kernel does not access any of the data structures defined in **sys/cfg/cxxx.c.** These structures are used only by the driver. They are provided in a separate file only so that the system administrator can change the configuration expected by the driver.

Driver Procedures

This section describes the routines in a simple character driver; most are called by the XENIX kernel; two are called by other driver routines. Aside from these routines, the file **sys/io/ixxx.c** must include the driver constants file **sys/h/ixxx.h** and also declare the data structures listed above as **extern** (external). For example, **ixxx.c** must reference the **ixxx_adr** array defined in **cxxx.c** and would declare it as follows:

```
extern unsigned ixxx_adr[];  /* array of port addresses */
```

ixxxinit Procedure

```
ixxxinit();
```

This procedure is optional but is normally provided for physical devices. It is called via the switch **dinitsw** during system initialization. **ixxxinit** should check each possible device handled by the driver to determine whether it is present in the system. For example, a driver that can handle up to four printers mapped to a range of I/O ports should check for the presence or absence of each printer. The checks can typically be done by writing a test pattern to a device register and then reading the device register for a response (if any). **ixxxinit** should write a line to the standard output for each device checked, indicating if it was found or not found. **ixxxinit** should initialize the static data structures for devices that are found, such as the output queue headers and the device status variables. **ixxxinit** should also initialize hardware to a known state, for devices that are found.

ixxxopen Procedure

```
ixxxopen(dev, oflag)
dev_t   dev;     /* device number, major/minor */
int     oflag;   /* flags specified to open system call, NOT USED */
```

ixxxopen is called by the kernel via **cdevsw** each time a device with the major number managed by the driver is opened. **dev** is the 16-bit device number. **oflag** contains the flags specified to the corresponding **open** system call (see **open** in "System Functions" in the *XENIX 286 C Library Guide*). These flags are typically not used by the driver **ixxxopen** routine.

ixxxopen should first validate its parameters. If the minor device number is out of the valid range, an error should be indicated. If the minor device number corresponds to a device not physically present, an error should be indicated. If the minor device number corresponds to a device that is already open and concurrent access is not allowed, an error should be indicated.

Errors can be indicated by assigning a nonzero error code to the kernel variable **u.u_error.** The kernel checks this variable when control returns to it from the device driver. The file **sys/h/user.h** predefines many error codes. These predefined error codes are described in the introduction to Appendix C, "System Functions," in the *XENIX 286 C Library Guide*. For example, to indicate a bad minor device number or one that refers to a device not physically present, use **EINVAL.** To indicate that a device is already open and cannot be concurrently opened again, use **EBUSY.**

If its parameters are valid, **ixxxopen** should enable device interrupts. (Device interrupts should be kept disabled when a device is not being used, to prevent device state changes from generating interrupts that the kernel and driver would have to handle.) **ixxxopen** should also assign the device status variable to indicate that the device is now open. In the case of a printer, **ixxxopen** may also write an initial character sequence to reset it and place the printer in a known state. The simplest such sequence is simply a formfeed, to place the printer at the top of form.

ixxxclose Procedure

```
ixxxclose(dev, oflag)
dev_t   dev;    /* device number, major/minor */
int     oflag;  /* flags specified to open system call, NOT USED */
```

ixxxclose is called by the kernel via **cdevsw** only for the *last* close of the device denoted by the device number **dev.** The kernel keeps track of the number of opens for each device without a matching close. It calls **ixxxclose** only for the last close that ends all activity on the device. For devices that do not permit concurrent access, this is the same as calling **ixxxclose** for every close operation.

oflag contains the flags specified to the **open** system call that created the file descriptor that was closed by the **close** system call that triggered the call to **ixxxclose.** These flags are typically not used by the driver **ixxxclose** routine.

ixxxclose should wait until all output in the queue is transferred to the device, assign the device status variable to indicate that the device is closed, and then disable device interrupts before returning.

ixxxread Procedure

```
ixxxread(dev)
dev_t   dev;    /* device number, major/minor */
```

ixxxread is not provided for write-only devices such as printers. In the **cdevsw** switch, the kernel procedure **nodev** takes the place of **ixxxread. nodev** assigns the value **ENODEV** to **u.u_error** if it is called. Chapter 4, "Terminal Drivers," includes a version of **ixxxread** for an example terminal driver.

ixxxwrite Procedure

```
ixxxwrite(dev)
dev_t    dev;       /* device number, major/minor */
```

ixxxwrite is called by the kernel via **cdevsw** when the user task makes the **write** system call. The number of characters to be written is specified by the **u** structure field **u.u_count.** The kernel routine **cpass** is called to transfer each character from the user task's address space. **cpass** also decrements **u.u_count.**

ixxxwrite is responsible for doing any conversion and checking required in the character stream. For example, **ixxxwrite** might expand tabs, replace illegal characters with a combination of printable characters, and translate end-of-line characters (linefeeds) to a carriage return/linefeed pair. The complexities of enqueuing a character (checking water marks, putting the task to sleep, etc.) should be isolated in a separate routine, **ixxxenq(unit, c).**

ixxxwrite can also check to see if an I/O error has been signaled by the interrupt handler; this is done by checking to see if the handler has set the device status to indicate an error. If this occurs, an error code should be assigned to **u.u_error** and no other action taken. **ixxxwrite** must do this on behalf of the interrupt routine because the interrupt routine cannot access the **u** structure.

Typical code for **ixxxwrite** is

```
{
int un;      /* unit (minor number)    */
int c;       /* char being transferred */

     if ((un = minor(dev)) >= ixxx_NUM)
          u.u_error = ENODEV;

     else if (ixxx_sts[un] == ixxx_ERR)
          u.u_error = EIO;

     else
          while (u.u_count) {
               c = cpass();
               /* Here is where possibly complex logic for conversion
                  goes, with multiple calls to ixxxenq to queue the
                  character(s) for output. */
          }
}
```

ixxxioctl Procedure

One other driver routine, **ixxxioctl,** is called via **cdevsw.** This routine is not present in the example simple character driver and is replaced in **cdevsw** by the kernel routine **nodev. nodev** assigns the value **ENODEV** to **u.u_error** if it is called. Chapters 4 and 5 contain more information about the **ixxxioctl** routine.

ixxxenq Procedure

```
ixxxenq(unit, c)
int    unit;    /* unit number = minor device number that's
                   been validated */
int    c;       /* char to be enqueued */
```

ixxxenq is called by **ixxxwrite** to enqueue a character on a device's output queue. It handles putting the task to sleep if the queue's high-water mark is reached and starting output by calling **ixxxstart** if the queue was empty. Typical code for **ixxxenq** is

```
{
        if (putc(c, ixxx_q[unit]) == -1 || ixxx_q[unit].c_cc >= ixxx_HIW)
            sleep(ixxx_q[unit], ixxx_PRI);

        if ((ixxx_q[unit].c_cc == 1) &&
            ~(inb(ixxx_adr[unit] + ixxx_STS) & ixxx_BSY))
            ixxxstart(unit);
}
```

ixxxstart Procedure

```
ixxxstart(unit)
int unit;       /* unit number = minor number that's been validated */
```

ixxxstart is called by either **ixxxintr** or **ixxxenq** to output a character from the output queue to the device. If the number of characters in the queue falls to the low-water mark, **ixxxstart** calls **wakeup** on the queue. **ixxxstart** does nothing if called for a device with an empty queue (**getc** returns -1). Typical code for **ixxxstart** is

```
{
int c;

        if ((c = getc(ixxx_q[unit]) >= 0)
            outb(ixxx_adr[unit] + ixxx_DAT, c);

        if (ixxx_q[unit].c_cc == ixxx_LOW)
            wakeup(ixxx_q[unit]);
}
```

ixxxintr Procedure

```
ixxxintr(level)
int level;    /* MULTIBUS interrupt level, in range 0-255 */
```

ixxxintr is called by the kernel for each interrupt that occurs for a device managed by the driver. The kernel fields all interrupts, determines the MULTIBUS interrupt level, and calls the **ixxxintr** procedure referenced by **vecintsw[level].**

ixxxintr should ignore extraneous interrupts from devices absent or not open; such interrupts may occur due to hardware problems. However, **ixxxintr** should acknowledge all interrupts as may be required by the device hardware.

ixxxintr should also check for errors. Errors should be indicated in the device status maintained by the driver. Note that **ixxxintr** cannot write to the variable **u.u_error** to indicate an error, as there is no guarantee that the associated task and its **u** structure will be swapped into memory when **ixxxintr** is called. Reporting errors via the device status requires cooperation with **ixxxwrite,** which must check the status and write the error code to the **u** structure if needed. In the example code, the hardware is presumed to give a simple indication of an unrecoverable "hard" error. Real drivers may have to handle and recover from various types of "soft" errors using a retry strategy.

Finally, if the unit is open, and the unit interrupted because a previous output has completed, and the unit did not encounter an error, **ixxxintr** should call **ixxxstart** to get the next character (if any) from the queue and write it to the device. Typical code for **ixxxintr** is

```
{
char  sts;
int   unit;

      /* loop through devices and handle those with interrupts to
         be acknowledged (ixxx_BSY set in status read from device) */
      for (unit = 0; unit < ixxx_NUM ; unit++) {

            /* Reading status port acknowledges interrupt
               and clears ixxx_BSY for subsequent reads. */
            sts = inb(ixxx_adr[unit] + ixxx_STS);

            if ((ixxx_sts[unit] == ixxx_OPN) &&
                  /* Ignore interrupt if unit not open. */
                  (sts & ixxx_BSY)) {
                  if (sts & (ixxx_BSY | ixxx_HER))
                        ixxx_sts[unit] = ixxx_ERR;
                  else
                        ixxxstart(unit);
            }
      }
}
```

Output Summary

Output of data using the driver begins with a kernel call to **ixxxwrite**, which checks for any I/O error reported by the interrupt routine, gets characters from the user task address space, does any needed conversions, and calls **ixxxenq** with the bytes to be output.

ixxxenq queues the characters in the **clist** structure and calls **sleep** if the queue fills to the high-water mark. **ixxxenq** also calls **ixxxstart** if the queue was empty before the current call to **ixxxenq**; this is to prime the interrupt-driven output cycle.

ixxxintr polls the devices handled by the driver, acknowledges interrupts, checks for I/O errors, and calls **ixxxstart** to output the next character from the queue for each active device. So long as a queue does not empty, each character output triggers a completion interrupt that causes the next character to be output, a self-sustaining cycle.

ixxxstart gets a character (if any) from a queue and calls the kernel routine **outb** to write the character to the device port. **ixxxstart** also wakes up any tasks waiting on the output queue if the queue reaches its low-water mark.

intel®

CHAPTER 4
TERMINAL DRIVERS

This chapter provides information about terminal drivers, a type of character device driver that handles interactive terminals or serial communications lines.

XENIX provides more supporting routines and data structures for terminals than for other character device drivers. Associated with each terminal is a **tty** structure defined by XENIX. Many terminal functions are handled by XENIX "line discipline" routines that handle device-independent input, output, and control functions in standard ways.

This chapter describes the **tty** structure, the line discipline routines provided, and then the components of a character driver. The final part of this chapter, "iSBC 534 Driver," is a listing of a terminal driver supplied as an example. This listing is only an example and is not the same as the iSBC 534 driver supplied with your XENIX 286 system.

The following material may be useful in conjunction with this chapter:

- Appendix B of this manual, "Converting Drivers from Release 1 to Release 3 of XENIX 286," includes tables that describe the **tty** fields and the input modes, output modes, control modes, and line discipline modes used with the **tty** structure.

- Appendix C of this manual, "tty.h Include File," lists the **tty.h** file that defines the **tty** structure and other information used by terminal drivers.

- Appendix D of this manual, "termio.h Include File," lists the **termio.h** file that defines the control characters, input modes, output modes, control modes, and line discipline modes used with terminal drivers.

- The entry **tty** in "Devices" in the *XENIX 286 Reference Manual* describes the general terminal interface and should be read in conjunction with this chapter.

- The entry **termcap** in "Files" in the *XENIX 286 Reference Manual* describes the terminal capabilities file **/etc/termcap** and should be read if you are adding a new kind of terminal to your XENIX 286 system. (Note that a new terminal device driver may be used with an existing type of terminal and not require any modification of the **termcap** file.)

- The entry **ttys** in "Files" in the *XENIX 286 Reference Manual* describes how a new terminal device can be added to the list of devices through which users can log in to the XENIX 286 system.

- The include file **sys/h/ttold.h** provides definitions used for backward compatibility with UNIX V7 terminal handling.

- The include file **sys/h/ioctl.h** defines identifiers for **ioctl** command codes.

tty Structure

A **tty** structure is defined for each terminal or communication line. The structure is defined in **tty.h**, which is listed in Appendix C of this manual. Each **tty** structure references three character queues. These queues are implemented using the **clist** structure, as described in Chapter 3, "Simple Character Drivers." The queues are the output queue, the raw input queue, and the canonical ("cooked") input queue. Character input is placed first in the raw input queue. The canonical input is what is normally seen by programs reading a terminal. It contains input after processing. For example, line editing functions such as backspace and kill-line have been handled before input is placed in the canonical queue. Character mapping (e.g., carriage return to linefeed) and character expansion (e.g., tab to blanks) may also occur between the two queues. Echoing of input characters is handled from the raw input queue. Characters written by the program are placed in the output queue. Any needed transformations (e.g., map tabs to blanks, linefeed to carriage return, linefeed) are done before characters are placed in the output queue. Input characters are copied to the output queue at interrupt time immediately after being read, to provide echoing of input.

The **t_proc** field references the driver-specific **ixxxproc** routine, which is called to perform device-dependent functions.

The **t_line** field is a small integer (in the range 0-127) that indexes the kernel's line discipline table, **linesw**, to select the line discipline routines used by the driver.

Other fields in the **tty** structure contain state information and other miscellaneous information used internally by the line discipline routines.

More information about the **tty** structure is contained in Appendix B, "Converting Device Drivers from Release 1 to Release 3 of XENIX 286," Appendix C, "tty.h Include File," and the entry **tty** in "Devices" in the *XENIX 286 Reference Manual.*

Line Discipline Routines

XENIX 286 handles most of the work of a terminal driver in a device-independent set of "line discipline" routines. These routines are accessed indirectly, via a switch table containing pointers to the routines. Use of the switch table enables a developer to add routines for a different line discipline if desired.

XENIX 286 Release 3 supports one set of line discipline routines, with index 0 in the **linesw** table in **c.c**. These routines are accessed via the **t_line** field in the driver's **tty** structure. Consider the following example:

```
struct tty ixxx_tty[ixxx_NUM];  /* declare tty structures for driver */
. . .
int unit = minor(dev);          /* dev is device number of type dev_t */
struct tty *tp = &ixxx_tty[unit];
. . .
/* Call the line discipline open routine. */
(*linesw[tp->t_line].l_open)(tp);
```

The following identifiers are used for offsets into a row of the **linesw** table:

l_open	the open routine
l_close	the close routine
l_read	the input routine
l_write	the output routine
l_ioctl	the control function routine
l_input	the input routine
l_output	the output routine

Note that the final entry in the **linesw** table in **c.c** and in the corresponding line of the **master** file is **nulldev,** a null "do-nothing" routine. This entry is for compatibility with older drivers that may still use that entry in the switch table. **c.c** and **master** are described in greater detail in Chapter 6, "Adding Devices to the Configuration."

A new set of line discipline routines can be added to XENIX 286 by inserting a line in the appropriate section (which is clearly labeled) of the **master** file. To access the new routines, a driver must modify the **t_line** field in its **tty** structure so that it indexes the row of **linesw** that specifies the new routines. The first entry in the line in **master** must name a character device that is also specified in the **master** file. For the standard line discipline, the device specified is **tty,** the general terminal interface. The next seven entries (separated by spaces) are the names of the routines for each of the functions listed above, open, close, etc. The final entry on the line is **nulldev.** For example, the line in **master** for the existing line discipline routines is

```
tty    ttopen ttclose ttread ttwrite ttioctl ttin ttout nulldev
```

The routines in this set are **ttopen, ttclose, ttread, ttwrite, ttioctl, ttin,** and **ttout.** These routines and other line discipline routines are described in the following sections. Some of the other line discipline routines are called directly by driver routines without going through the line discipline switch **linesw;** this is reasonable for routines that are unlikely to change from one line discipline to another. One of the routines described, **ttxput,** is internal to the line discipline and never called by the driver, but is described to clarify how output is handled by the line discipline routines.

ttinit

```
ttinit(tp)
struct tty  *tp;    /* tty structure for device */
```

ttinit is called directly (not via **linesw**) by the driver **ixxxopen** routine to initialize the **tty** structure for the device. **ttinit** is only called by **ixxxopen** if the device was not already open. **ttinit** is called before **ttopen,** as described below in the section "**ixxxopen**".

ttopen

```
ttopen(dev, tp)
dev_t       dev;        /* device number, major/minor   */
struct tty  *tp;        /* tty structure for device     */
```

ttopen is called by the driver's **ixxxopen** routine. **ttopen** initializes the three queues and other fields in the **tty** structure. **ttopen** is only called by **ixxxopen** if the device was not already open. **ttopen** is called after **ttinit,** as described below in the section "**ixxxopen**".

ttclose

```
ttclose(tp)
struct tty  *tp;        /* tty structure for device     */
```

ttclose is called by the driver's **ixxxclose** routine. **ttclose** flushes the input queues in the **tty** structure, waits for any output to complete, and then assigns relevant fields in the structure to indicate that the device is closed.

ttread

```
ttread(tp)
struct tty  *tp;        /* tty structure for device     */
```

ttread is called by the driver's **ixxxread** routine and handles all aspects of the **ixxxread** call (and of the **read** system call that triggered the call to **ixxxread**). **ttread** gets its data from the canonical input queue (unless the terminal is in a special "raw" input mode) and waits for more input if necessary. **ttread** transfers data to the calling process using the address and count found in the process's **u** structure. While the calling process can request any number of characters, at most one line is returned. If a complete line is returned, then the last character read by the request is the newline (ASCII linefeed).

ttwrite

```
ttwrite(tp)
struct tty  *tp;        /* tty structure for device     */
```

ttwrite is called by the driver's **ixxxwrite** routine and handles all aspects of the **ixxxwrite** call (and of the **write** system call that triggered the call to **ixxxwrite**). **ttwrite** transfers data from the calling process using the address and count found in the process's **u** structure. **ttwrite** calls the internal line discipline routine **ttxput** to place a character in the output queue and perform any needed character expansion (e.g., add delay characters, expand tabs). **ttwrite** guards the high-water mark for the output queue and suspends the calling process by calling **sleep** if the mark is reached.

ttiocom

```
ttiocom(tp, cmd, addr, dev)
struct tty    *tp;      /* tty structure for device          */
int            cmd;     /* command code                      */
faddr_t        addr;    /* pointer to structure with command arguments */
dev_t          dev;     /* device number, major/minor        */
```

ttiocom is called directly (not via **linesw**) by **ixxxioctl,** which is called when an **ioctl** system call is made for the device. **ttiocom** handles various device control functions. **ttiocom** returns zero if no further driver action is required and nonzero if the driver must reconfigure the device by calling **ixxxparam.** An example of a control function that requires device reconfiguration is a change in baud rate, which normally requires that the new rate be communicated to the device hardware. **ttiocom** calls the line discipline routine **ttioctl** (if line discipline 0 is used) to handle line-discipline-dependent parts of the device control functions.

The different **ioctl** commands and the format of the command arguments for terminals are described in the entry **tty** in "Devices" in the *XENIX 286 Reference Manual.* The include file **sys/h/ioctl.h** defines the identifiers used for **ioctl** command codes.

ttioctl

```
ttioctl(cmd, tp, addr, dev)
int            cmd;     /* command code                      */
struct tty    *tp;      /* tty structure for device          */
caddr_t        addr;    /* pointer to structure with
                           command arguments                 */
dev_t          dev;     /* device number, major/minor        */
```

ttioctl is called (via **linesw**) by the **ttiocom** routine to handle line-discipline-dependent parts of the I/O control functions. **ttioctl** is not directly called by driver routines even though it is one of the routines listed in **linesw.**

ttin

```
ttin(c, tp)
char           c;       /* character to be input             */
struct tty    *tp;      /* tty structure for device          */
```

ttin is called by the driver interrupt routine when the driver receives a character from the device. **ttin** places the character in the raw input queue and calls **ttxput** to echo each character (if echo is enabled) by placing it in the output queue.

ttout

```
ttout(tp)
struct tty    *tp;      /* tty structure for device          */
```

ttout is called to start output when characters are on the output queue. **ttout** is called from the driver interrupt routine when handling a transmitter ready interrupt. **ttout** calls **ixxxproc** to actually output characters.

ttxput

```
ttxput(c, tp)
char              c;    /* character to be output            */
struct tty    *tp;      /* tty structure for device          */
```

ttxput is not called directly by the driver but is a routine internal to the line discipline. **ttxput** is called by **ttwrite** and also by **ttin** (for echoing) to place a character in the output queue. **ttxput** handles any needed character expansion (e.g., expanding tabs to spaces, inserting delay characters).

Modem Control by Terminal Drivers

Dial-in modem lines are among the devices controlled with terminal drivers. If bit 6 of a terminal device minor number is 1, then the device is being used as a dial-in line and the driver should provide modem control. Bits 0-5 of the minor number should contain the normal minor number of the device. Bit 7 should be 0 and never used by the driver; bit 7 of terminal device minor numbers is reserved for future use by Intel. For example, consider a line that is used sometimes as a dial-out line and sometimes as a dial-in line. The line would be accessed via one device special file as a dial-out line, with a minor number with bit 6 clear, e.g., 17. The line would be accessed via a second device special file as a dial-in line, with a minor number with bit 6 set, e.g., 81 = 17 + 64.

Driver Description

This section describes terminal driver routines. It does not repeat information on driver organization or declarations covered in Chapter 3, "Simple Character Drivers." You will want to consult the driver example, in the section "iSBC 534 driver" at the end of this chapter, while reading this section. Each of the steps described is actually implemented by the corresponding iSBC 534 driver routine. For example, code for the steps given for the **ixxxopen** routine can be found in the **i534open** routine. With the help of the descriptions in this section, you should be able to distinguish the general-purpose code in the iSBC 534 driver from device-specific code. Of course, much of the code in the driver example is device-specific, simply because the kernel line discipline routines do much of the device-independent work of terminal handling.

Your reading of the example code should focus on the last file, **sys/io/i534.c.** The include file, **sys/h/i534.h,** is almost entirely device-specific definitions, except for **SPL** and **MINORMSK.** The configuration and data structures file, **sys/cfg/c534.c** should be understood; it is only a single page and easily grasped. The configuration parameters are the number of boards allowed and an array of base port addresses for the boards. Each board contains four lines (four devices). Data structures include an array of **tty** structures (one per device), an array of base port addresses for each device, an array indicating which boards are present, and an array that records the current baud rate for each device.

ixxxinit Procedure

```
ixxxinit();
```

This procedure is optional but is normally provided for physical devices. It is called via the switch **dinitsw** during system initialization. **ixxxinit** checks each possible board handled by the driver to determine if it is present or absent. **ixxxinit** writes a message to the system console for each board saying whether the board was found or not. (The kernel's version of **printf** writes directly to the system console.) **ixxxinit** initializes a board status variable that indicates whether the board is present or absent. For boards that are present, **ixxxinit** initializes device hardware. Note that in the iSBC 534 driver, the responsibilities of the **i534init** routine are partly handled by the **i534check** routine, called by **i534init.**

ixxxparam Procedure

```
ixxxparam(dev)
dev_t       dev;       /* device number, major/minor    */
```

ixxxparam is called whenever the device hardware must be configured or reconfigured to adapt to requested line options. **ixxxparam** is called from **ixxxopen** to configure the line when it is opened and is called from **ixxxioctl** for reconfiguration. The **tty** structure must be initialized (by **ttinit**) and have the fields **t_proc, t_oflag, t_iflag, t_lflag,** and **t_cflag** assigned to their desired values before **ixxxparam** is called. An example of what **ixxxparam** does is communicating the requested baud rate to the hardware.

The code in **i534param** is almost entirely device-specific but some parts are of general interest. If a baud rate is requested that is not valid for the device, then **u.u_error** is assigned the error code **EINVAL** and **i534param** returns. If a baud rate of 0 is specified and the device is a dial-out line (bit 6 set in the minor number), then **i534param** turns off Data Terminal Ready, causing the modem to hang up.

ixxxopen Procedure

```
ixxxopen(dev, oflag)
dev_t   dev;    /* device number, major/minor */
int     oflag;  /* flags specified to open system call, NOT USED */
```

ixxxopen is called each time a device managed by the driver is opened. **ixxxopen** first checks that the unit can be opened--the minor number is valid and the device is present. Otherwise an error code is assigned to **u.u_error** and **ixxxopen** returns. If no error was detected:

1. If this is the first open of this device (the **ISOPEN** bit in the **tty** structure is zero) then do the following:

   a. Initialize the **tty** structure. Fill in the **t_proc** field with the address of the **ixxxproc** routine. Then call **ttinit** with the address of the **tty** structure to be initialized. Then initialize the mode flags in the structure: output modes, input modes, line discipline modes, and control modes.

   b. Call the **ixxxparam** routine for the device to make any necessary changes to the device control registers as specified in the mode flags in the **tty** structure.

   c. If modem control is enabled for the device (bit 6 in the minor number is set), then set up the device hardware for an incoming call and call **sleep**, waiting to be awakened when the device has received a call and has Carrier Detect.

2. Otherwise (if this is not the first open), check the **XCLUDE** bit in the **t_lflag** field of the **tty** structure. If this bit is set, concurrent access to the device is not allowed except by the super-user (**u.u_uid == 0**). If the bit is set and the caller is not the super-user, assign the error code **EBUSY** to **u.u_error** and return.

3. Set the **CARR_ON** bit in the **t_state** field of the **tty** structure (even if the device is not using modem control).

4. Call the line discipline open routine to do device-independent open processing.

All of these steps are illustrated by the **i534open** routine.

ixxxclose Procedure

```
ixxxclose(dev, oflag)
dev_t   dev;    /* device number, major/minor */
int     oflag;  /* flags specified to open system call, NOT USED */
```

ixxxclose is called on the last close of a device. **ixxxclose** calls the line discipline close routine to do device-independent close processing, which includes discarding any unconsumed input and waiting for any unwritten output to the device to complete. **ixxxclose** then turns off Data Terminal Ready on the device (for modem control, if modem control is supported) and may perform other device-dependent functions, such as clearing the transmitter and receiver registers of the device. These steps are illustrated by the **i534close** routine.

ixxxread Procedure

```
ixxxread(dev)
dev_t       dev;       /* device number, major/minor   */
```

This procedure is called to handle the **read** system call and simply calls the read routine in the line discipline. An example is the **i534read** routine.

ixxxwrite Procedure

```
ixxxwrite(dev)
dev_t       dev;       /* device number, major/minor   */
```

This procedure is called to handle the **write** system call and simply calls the write routine in the line discipline. An example is the **i534write** routine.

ixxxintr Procedure

```
ixxxintr(level)
int         level;     /* interrupt level, from PIC, in range 0-71 */
```

ixxxintr is called by the kernel for each interrupt from a device managed by the driver. For the iSBC 534, the boards share a single interrupt level, requiring that the boards be polled for their interrupt status. However, each board serializes interrupts from that board, and the status read from the board indicates which device is associated with the particular interrupt. **i534intr** polls all the boards repeatedly until a pass is made with no interrupts found. **i534intr** handles three kinds of interrupts: receiver interrupts (a character has been received), transmitter interrupts (a transmitter is ready for another character), and modem interrupts (a dial-in line connected or was hung up).

For a receiver interrupt, **i534intr** reads the character received. If the device is not open, the character is ignored and discarded. Otherwise, the device status is checked for read errors; any read error sets one or more indicator bits in the upper bits of the 16-bit value that holds the received character. The line discipline input routine is then called to handle enqueuing the character.

For a transmitter ready interrupt, **i534intr** first clears the **BUSY** flag in the **t_state** field of the device's **tty** structure. **i534start** is then called to send the next character. (Note: Normally **ixxxintr** should call **ixxxproc** to send the next character. **i534intr** calls **i534start** directly as an optimization.) On returning from **i534start, i534intr** checks to see if any processes are waiting on the device output queue (**OASLP** set in the **t_state** field) and if the number of characters in the output queue is less than or equal to the low-water mark for the baud rate being used. If both conditions are satisfied, the **OASLP** bit is cleared in **t_state** and **wakeup** is called to rouse the sleeping process(es).

For a modem interrupt, **i534intr** checks a different device register. A hangup interrupt is ignored unless the **t_state** field indicates that the device was open and that the carrier was on. For a valid hangup, the **SIGHUP** signal is sent to all processes in the process group associated with the device. Data Terminal Ready is then turned off to cause the modem to hang up at its end of the line. Carrier Detect is then also turned off. If a ring interrupt occurs, then the process sleeping (in **i534open**) waiting for Carrier Detect is awakened.

ixxxproc Procedure

```
ixxxproc(tp, cmd)
struct tty   *tp;      /* tty structure for device      */
int           cmd;     /* command code                  */
```

ixxxproc is called whenever a change must be made in the device's output. The command codes that **ixxxproc** must handle are defined in **tty.h.** The following list gives all the commands along with their meaning and how they are handled:

T_OUTPUT Start output; simply call **ixxxstart.**

T_TIME Time delay has finished; clear the **TIMEOUT** bit in the **t_state** field of the **tty** structure and call **ixxxstart.**

T_SUSPEND Suspend output on this line (e.g., CONTROL-S received); set the **TTSTOP** bit and clear the **BUSY** bit in the **t_state** field of the **tty** structure. Note that the suspension takes effect immediately and applies to characters that are already in the output queue.

T_RESUME Resume output on the line (e.g., CONTROL-Q received); clear the **TTSTOP** bit in the **t_state** field of the **tty** structure and call **ixxxstart.**

T_BLOCK Send a stop character, which should block future input (perhaps after a lag); if the stop character is successfully queued, set the **TBLOCK** bit in the **t_state** field of the **tty** structure and call **ixxxstart.**

T_UNBLOCK Send a start character, which should resume input (perhaps after a lag); if the start character is successfully queued, clear the **TBLOCK** bit in the **t_state** field and call **ixxxstart.**

T_RFLUSH Command to flush the input queue; this command is handled by other line discipline routines before calling **ixxxproc,** so do nothing and return.

T_WFLUSH Command to flush the output queue; handle like **T_RESUME,** ensuring that **TTSTOP** is clear and that output is started; control returns without waiting for the queue to be emptied.

T_BREAK Send a "break", a sequence of zero bits lasting approximately 1/4 of a second of real time.

ixxxstart Procedure

```
ixxxstart(tp)
struct tty    *tp;      /* tty structure for device        */
```

ixxxstart is called by **ixxxproc** to start output for a device. **i534start** locks out character interrupts until it has set the **BUSY** flag that guarantees that the remaining code in the routine will not be re-entered. While interrupts are locked out, **i534start** checks the **BUSY, TIMEOUT,** and **TTSTOP** flags in the **t_state** field of the **tty** structure. If any of the three flags is set, **i534start** restores the previous interrupt priority and returns. Otherwise, **i534start** sets the **BUSY** flag itself, restores the previous interrupt priority, and starts output. Note the distinction between "raw" output (no postprocessing) and cooked output (postprocessing to insert timeouts in **i534start** and other processing in previous routines). The final code in **i534start** awakens (first) any processes waiting for the output queue to drain and (second) any processes waiting for the low-water mark on the output queue.

ixxxioctl Procedure

```
ixxxioctl(dev, cmd, addr, oflag)
dev_t       dev;      /* device number, major/minor   */
int         cmd;      /* command code                 */
faddr_t     addr;     /* pointer to structure with command arguments   */
int         oflag;    /* flags specified to open system call, NOT USED   */
```

ixxxioctl is called when an **ioctl** system call is made for the device. The different **ioctl** commands and the format of the command arguments for terminals are described in the entry **tty** in "Devices" in the *XENIX 286 Reference Manual*. The include file **sys/h/ioctl.h** defines the identifiers used for **ioctl** command codes.

i534ioctl simply calls the line discipline routine **ttiocom** directly (not via **linesw**) to handle all the **ioctl** commands. **ttiocom** returns zero if no further driver action is required and nonzero if the driver must reconfigure the device. If a nonzero value is returned by **ttiocom, i534ioctl** calls **i534param** to reconfigure the device.

iSBC® 534 Driver

This section lists source code for Intel's XENIX 286 terminal driver for the iSBC 534 Four Channel Communications Expansion Board. This code may not correspond to the latest version of this driver supplied with your XENIX 286 system; this code is included here only as an example of a terminal driver. There are three source files in the driver: **sys/h/i534.h** (include file), **sys/cfg/c534.c** (data structures), and **sys/io/i534.c** (routines).

sys/h/i534.h Listing

```
#define     SPL         spl5                /* keep interrupts away */

#define     ISPEED              13      /* initial baud rate of 9600 = = (13);300 = = (7)*/
#define     MINORMSK            0x1F    /* reserve bit 7 ; bit 6 for modem */
#define     MODEMMSK            0xC0    /* bit 6 of the minor number sets modem op */
#define     DTRON               0x80    /* bit 8 of usart status byte; 1 if present */

/*
 * Structures for the iSBC534
 *
 * ------------------------------------------------------------
 * Commands used for operation and initialization of
 * pic's,pit's,usarts and the ppi
 *
 *
 * ------------------------------------------------------------
 * Written by J. Chorn                      December 7th 1981
 *
 * MODIFICATION HISTORY
 *      1001        llk         1/25/84
 *          Ported to system 3 UNIX.
 *
 *
 * ------------------------------------------------------------
 * Refer to iSBC534 Hardware Reference Manual Chapter Three I/O Address
 * Assignments for further information.
 *
 * Device PHYSICAL port layout
 * Based on Data Block Select
 *
 *
 * Usart I/O functions:
 *
 *          Write: I/O
 *          Read:  I/O
```

```
*
* PIC status port functions:
*
*       Write:  ICW1,OCW2, and OCW3
*       Read:   Status and Poll
*
* PIC mask port functions:
*
*       Write:  ICW2 and OCW1(mask)
*       Read:   OCW1
*
*
* structure db534:
*
*       8251 Usarts     [4]             {2 bytes wide each}
*       8259 PIC's      [2]             {2 bytes wide each}
*       Board command port's            {4 bytes}
*
*/

struct db534{
        struct  {
                char    data;                   /* serial I/O data port */
                char    cntrl;                  /* serial control port */
        } USART[4];
        struct  {
                char    csr;                    /* PIC status port */
                char    msr;                    /* PIC mask port */
        } PIC[2];
        char    selcntr;                        /* Select control block */
        char    seldata;                        /* Select data block */
        char    stestmd;                        /* Select/deselect test mode */
        char    reset;                          /* Board Reset port */
};

/*
 * Refer to iSBC534 Hardware Reference Manual Chapter Three I/O Address
 * Assignments for further information.
 *
 * Device Physical Port Layout
 * Based on Control Block Select
 *
 *
 * 8253 PIT Functions:
 *
 *       Load/Read count
 *
 * 8255 PPI Functions:
 *
 * Write :
 *       Port A None
 *       Port B None
 *       Port C Data out
 *       Port (Control) Control commands
```

```
*
* Read :
*
*       Port A Data in
*       Port B Data in
*       Port C Data status
*       Port (Control) None
*
*
* structure cd534:
*
*       8253 PIT's [2]                          {4 byte wide each}
*       8255 PPI Parallel port                  {4 bytes}
*       Board command port's                    {4 bytes}
*
*/

struct cb534{
        struct          {
                        char            timer[3];       /* read/load BDG[?] & BDG[? + 3] */
                        char            pcr;            /* PIT control register */
        } PIT[2];
        char            porta;                          /* read port a data in */
        char            portb;                          /* read port b data in */
        char            portc;                          /* read/write port c */
        char            ppi_pcr;                        /* PPI control register */
        char            selcntr;                        /* select control block */
        char            seldata;                        /* select data block */
        char            stestmd;                        /* select/deselect test mode */
        char            reset;                          /* board reset */
};

/*
 * Structure for base assignments of boards in the configuration
 * This is used to associate the board with the base address
 */
struct i534cfg{
        int             c_base;
};


/*
 *
 * 8253 PIT commands
 *
 *      RATEMD0:        read/load timer0(or 4) for mode 3 (baud rate generator)
 *
 *      U534SPEED:              int constant of 1600 pit count in hex.
 *
 *      NOTE:           mode zero can only be used on timers 4 or 5 !! (according to
 *                      the manual ) and this is garf. Program for mode 3 for two
 *                      second clock signal.
 *
 */
```

```
#define     RATEMD0             0x36
#define     U534SPEED           0x0640

/*
 * 8251     USART command instructions
 *          I001    split instructions into separate bits
 */

#define     S_TXEN      0x01            /* transmitter enable */
#define     S_DTR       0x02            /* data terminal ready */
#define     S_RXEN      0x04            /* receiver enable */
#define     S_SBRK      0x08            /* send break char */
#define     S_ER        0x10            /* error reset */
#define     S_RTS       0x20            /* request to send */
#define     S_IR        0x40            /* internal reset */


#define     S_RXRDY             0x02            /* receiver has data */
#define     S_TXRDY             0x01            /* transmitter empty */
#define     S_PERROR            0x08            /* parity error */
#define     S_FRERROR           0x20            /* framing error */
#define     S_OVERRUN           0x10            /* overrun */

/*
 * 8251     USART mode instructions
 *          I001    split instructions into separate bits
 */

#define     S_BAUDF             0x02    /* baud rate factor = 16x */
#define     S_5BPC              0x00    /* 5 bits per char */
#define     S_6BPC              0x04    /* 6 bits per char */
#define     S_7BPC              0x08    /* 7 bits per char */
#define     S_8BPC              0x0C    /* 8 bits per char */
#define     S_PAREN             0x10    /* parity enable */
#define     S_PAREVEN           0x20    /* even parity */
#define     S_1STOP             0x40    /* 1 stop bit */
#define     S_2STOP             0xC0    /* 2 stop bits */

/*
 * 8259     PIC commands
 *
 * ICW'S
 *
 * PICICW1:             Format = 4,single pic,edge triggered.
 * PICICW2:             Initialization address.
 *                      Picicw3 and picicw4 not needed.
```

```
* OCW's
*
* MASKINT:          Set 8259 to mask all interrupt levels.
*
* GETINT :          Set 8259 for polled mode
*                   read requesting device on next rd pulse.
* GOODINT:          Mask to check bit 7 ;produce from the 'getint' command.
*                   Bit 7 set = = valid interrupt at this pic.
*
* TIMERGO:          Pic mask which allows bdg5 timer interrupts
*
*/

#define   PICICW1   0x16
#define   PICICW2   0
#define   MASKINT   0xFF
#define   GETINT    0x0C
#define   GOODINT   0x80
#define   TIMERGO   0xFD
```

sys/cfg/c534.c Listing

```
/*
 * c534.c
 *          iSBC 534 Specific Configuration file.
 *
 * This split out from c.c to avoid name-clashing with other device-
 * specific configuration files.
 */

#include "../h/param.h"                 /* this include types.h */
#include "../h/tty.h"
#include "../h/i534.h"


/*
 * N534 must be modified if the configuration
 * changes the number of isbc534 boards in the system.
 *    I001          moved this here from i534.h to consolidate configuration
 *                  options to the appropriate place.
 */
#define NUM534          2               /* Number of isbc534's in configuration  */
int N534 = NUM534;


/*
 * I001  moved these declarations to here from i534.c
 */
struct tty i534tty[NUM534*4];                           /* 4 USARTs per 534 */
short  i534addr[NUM534*4];                              /* parallel to tty struct */
int         i534alive[NUM534];                          /* does it live ?? */
int         i534speed[NUM534*4];                        /* current speed of tty */


/*
 * This table gives the interrupt level and board-base I/O address
 * for each possible iSBC 534 controller.  The driver procedure entry-
 * points are configured in c.c
 * To reconfigure for a different number of 534's, you must add or
 * delete the appropriate addresses in the structure below.
 */

struct     i534cfg    i534cfg[NUM534]   = {
                      0x30,             /* first board base addr = 0x30 */
                      0x40
};
```

sys/io/i534.c Listing

```
/*
 * isbc534 device driver.
 *
 * This is the set of procedures that make up the isbc534 device driver.
 * The procedures provided include i534open, i534close, i534intr, i534proc,
 * i534ioctl, i534read, and i534write which are the interfaces between
 * xenix and the hardware.  The subroutines used are i534init, i534param,
 * i534start, which are used to program the hardware. The isbc534 hardware
 * consists of 4 usarts, 2 pic's, 2 pit's and a ppi.
 * The ppi is not supported by this driver.
 *
 * Multiple isbc534 minor number structure:
 * bits 0-4:
 * Minor #:          Board:
 * 0-3  usarts       1st Board lowest intr level
 * 4-7  usarts       2nd Board next lowest intr level
 *
 * 12-15 usarts      4th Board last intr level
 *
 * NOTES:    The base address of the board MUST be non-zero!!!
 *           The isbc86/12 board must have the fail safe timer
 *           installed.(default)
 *
 *           The isbc534 REQUIRES a HARDWARE MODIFICATION for MODEM SUPPORT
 *                The isbc534 requires a default jumper removed from
 *                     pin 105-106
 *                and add a jumper from
 *                     pin 105-104
 *                This modification cascades timer bdg4 to bdg5 to allow a
 *                2 second timer used in detecting carrier from a modem.
 *                The carrier loss signal is generated via a separate
 *                interrupt.
 *           The above modification is ONLY NEEDED FOR MODEM SUPPORT but should
 *           be done for consistency.
 *
 *
 *           Debug switches are:  DEBUG for isbc534 support.
 *                                i534debug: output control
 *                                     0 = = synchronous routine tracing
 *                                     1 = = interrupt tracing
 *                                     2 = = all output
 *
 * Written by Jim Chorn
 *           on 12/29/81
 *
```

```
* History:    modified 1/15/82 for multiple board support.
*             modified 1/29/82 for console support.
*             modified 3/29/82 for addition of modem support
*                   mods affect i534open,i534close,i534intr.
*             modified 4/22/82   moved console support out to support isbx351
*             modified 6/22/82 added OR tie'ng of 534's on the same interrupt
*                   level.
*                   Changed the modem support bit to 0xC0 meaning configure
*                   the line for detection of aquisition AND loss of carrier
*                   detect signal. Bit 0x40 means detection of aquisition and
*                   bit 0x80 means detection of loss of carrier detect signal.
*                   The detection of aquisition of carrier without detection of
*                   loss of carrier is meaningless and is not mentioned in the
*                   manual entry.
*       6/28/83     plb   I000
*                   added fix to race condition
*       1/27/84     llk   I001
*                   ported to system 3 unix
*       2/6/84      llk   I002
*                   attempt to set line speed to 0 via stty now returns error
*
*/

#include  "../h/param.h"
#include  "../h/conf.h"                 /* system configuration */
#include  "../h/dir.h"                  /* system directory structures */
#include  "../h/a.out.h"                /* needed for user.h */
#include  "../h/user.h"                 /* user structures (system)   */
#include  "../h/tty.h"                  /* device structures (system)  */
#include  "../h/ioctl.h"                /* ioctl commands */
#include  "../h/i8259.h"                /* some pic commands from system */
#include  "../h/usart.h"                /* baud rates */
#include  "../h/i534.h"             /*   hardware structure and local commands */

#ifdef DEBUG
int         i534debug = 0;              /* debug output control */
#endif

extern      int         N534;                   /* number of boards configured in */
extern      struct tty i534tty[];               /* 4 USARTs per 534 */
extern      short i534addr[];                   /* device addrs for each tty struct */
extern      struct      i534cfg i534cfg[];      /* board software addresses von conf*/
extern      int         i534alive[];            /* does it live ?? */
int                     i534wakeup;             /* wakeup variable for modems */
```

```
/*
* This procedure verifies that a isbc534 board is presently
* configured by putting the board into test mode and
* then checking if the board actually is in test mode.
* This test mode check is a one bit test. If the board configured is not
* present an array variable for each board called i534alive is set to
* false.
* I001          changed the name of this routine from i534probe to i534check
*               to avoid conflict with i534proc
*
* TITLE:        i534check
*
* CALL:         i534check();
*
* INTERFACES:   i534init
*
* CALLS:        none
*
* History:
*
*/

i534check()
{
        register        board;
        register struct i534cfg *cf;
        struct db534    *DBbase;    /* set up the i/o boards base address */
        int             alive;

        for (board = 0; board<N534; board + +){
                cf = &i534cfg[board];
                if(cf->c_base ! = 0) {
                        alive = 1;                      /* assume it lives */
                        DBbase = (struct db534 *) cf->c_base;
                        outb(&DBbase->stestmd, 1);    /* select test mode */
                        if((inb(&DBbase->stestmd) & 1) = = 0 )
                        /* is test mode selected?          */
                                alive = 0; /* trash base addr for intr() */
                        outb(&DBbase->stestmd, 0xff);
                        if((inb(&DBbase->stestmd) & 1) = = 0)
                                alive = 0;
                        outb(&DBbase->stestmd, 0);    /* deselect test mode */
                        printf("iSBC 534 Based %x board %d %s.\n",
                                cf->c_base, board,
                                alive ? "found" : "NOT found" );
                        i534alive[board] = alive;
                }
        }
}
```

```
/*
* This procedure initializes the isbc534 when the call to dinit is
* made. This procedure is done ONCE ONLY in the following sequence:
*           initialize the isbc534 structures to point at the board,
*           reset the board,
*           initialize and mask the on-board pic's.
* After this has been accomplished there is no reason to reinitialize these
* functions on the isbc534 except when hardware failure occurs.
* NOTE: The baud rate clocks are not programmed here; this
* is done on device open and ioctl in the call to i534param.
* Same is true for the usart initialization.
*
* TITLE:            i534init
*
* CALL:             i534init();
*
* INTERFACES:       dinit
*
* CALLS:            i534check, outb
*
* History:    1/11/82   Shortened the delay time from 100 to 10 to speed things
*                       up a bit.
*             1/15/82   Added probing for boards.
*         l001      moved init of PIT's and usarts to open routine
*
*/

i534init()
{
          register struct db534    *DBbase;     /* set up i/o boards base addr */
          register int board;
          int i;

#ifdef DEBUG
          if(i534debug = = 0   ||   i534debug = = 2)
                printf("i534init, ");
#endif
          i534check();
          for(board = 0; board < N534; board + + ) {
                if(i534alive[board] = = 0)
                      continue;     /* Board not there!     */
                DBbase = (struct db534 *)  i534cfg[board].c_base;
                outb(&DBbase->reset, 0);
                outb(&DBbase->seldata, 0);
```

```
                for (i = 0;i<4;i + +) {
                        /* do the mystical h/w reset 0x80,0 */
                        outb(&DBbase->USART[i].cntrl,0x80);
                        ddelay(10);
                        outb(&DBbase->USART[i].cntrl,0);
                        ddelay(10);
                        /* l001: now do the reset mentioned on the 8251
                                data sheet */
                        outb(&DBbase->USART[i].cntrl,0);
                        ddelay(10);
                        outb(&DBbase->USART[i].cntrl,0);
                        ddelay(10);
                        outb(&DBbase->USART[i].cntrl,0);
                }
                outb(&DBbase->PIC[0].csr,   PICICW1);
                outb(&DBbase->PIC[0].msr,   PICICW2);
                outb(&DBbase->PIC[1].csr,   PICICW1);
                outb(&DBbase->PIC[1].msr,   PICICW2);
                outb(&DBbase->PIC[0].msr,   MASKINT);       /* mask all intrs */
                outb(&DBbase->PIC[1].msr,   MASKINT);
        }
}


/*
 * This procedure sets up a usart timer for a load operation and
 * programs the parameters of the line.  The code depends on having the
 * tty structure filled out before a call is made
 * to i534param. This is the sequence of events;
 *              check for valid speed
 *              program timer       (using i53tprog)
 *              program for the desired paramters
 * This procedure will program bdg0 to bdg4 as a baud rate generator.
 *
 *
 * TITLE:       i534param
 *
 * CALL:        i534param(dev);
 *
 * INTERFACES:      i534open
 *
 * CALLS:       i53tprog
 *
 * History:     1/20/82     :   removed bdg4, bdg5 programing options.
 *                              These timers aren't used.
 *              1/29/82     :   added console programming
 *              4/7/82      :   added i53tprog to handle pit programming
 *              4/22/82     :   removed console programming
 *
 *   l001       llk     1/27/84
 *              added support for parameter change
 *
```

```
 */

#define MAXBAUDS 15          /* maximum indexes into i534baud[] */
int i534baud[] = {
            US_B0,           US_B50,          US_B75,          US_B110,     0,
            US_B150,         US_B200,         US_B300,         US_B600,     US_B1200,
            0,               US_B2400,        US_B4800,        US_B9600,          0,
            0
};
extern int i534speed[];                       /* track record of baud rate */

i534param(dev)
int dev;
{
        struct cb534        *CBbase;          /* set up the i/o boards base address */
        struct db534        *DBbase;          /* set up the i/o boards base address */
        register struct tty *tp;
        register s;                           /* speed, mutex, etc */
        int port;                             /* usart and pit control ports */
        int picport;                          /* pic control port */
        int mask;                             /* intr mask for pic */
        int f;                                /* control mode flag from tty struct */
        mode;                                 /* mode to program 8251,8253 into */
        int unit, speed;
        short taddr;

        unit = dev & MINORMSK;                /* unit = board number */
#ifdef DEBUG
        if(i534debug = = 0 || i534debug = = 2)
                printf("i534param unit = %d, ",unit);
#endif
        tp = (struct tty *) &i534tty[unit];
        taddr = i534addr[unit];
        DBbase = (struct db534 *) (i534cfg[unit >> 2].c_base);
           /* beginning of block */
        CBbase = (struct cb534 *) (i534cfg[unit >> 2].c_base);
           /* beginning of block */
```

```
s = (int)tp->t_cflag & CBAUD;          /* s <- new requested speed */
if(s = = 0)    {                       /* hangup signal via stty */
        if (minor(dev) & MODEMMSK) {
                /*
                 * flick dtr off to cause hardware
                 * hang up on modem
                 */
                while ((inb(&DBbase->USART[taddr&03].cntrl) &
                    S_TXRDY) = = 0)
                ;                                   /* wait for txrdy */
                outb(&DBbase->USART[taddr&03].cntrl, S_ER);
                 /* dtr off */
        }
        else {                                  /* not a modem--illegal speed */
                u.u_error = EINVAL;             /* I002 */
                return;
        }
}
if(s ! = i534speed[unit]) {                     /* change speed? */
        i534speed[unit] = s;
        speed = i534baud[s];
        if ((s > MAXBAUDS) || ((s ! = 0) && (speed = = 0))) {
                u.u_error = EINVAL;             /* invalid baud rate */
                return;
        }
        unit % = 4;                             /* which usart? */
        if (unit = = 3){
                port = (int) &CBbase->PIT[1].timer[0];
                mode = RATEMD0;
        }else{
                port = (int) &CBbase->PIT[0].timer[unit];
                mode = RATEMD0 | (unit <<6);
        }
        s = SPL();
        outb(&CBbase->selcntr,1);
        i53tprog(port,(port|0x03),mode,speed);
        outb(&CBbase->seldata,1);
        splx(s);
}
```

```
        else
                unit %= 4;                              /* unit == port num of board */

        /*
         * I001
         * set parameters of line
         * first, set up the mode variable from the tty structure info.
         * then reset the usart and program the new mode.
         */
        f = tp-> t_cflag;
        mode = S_BAUDF | ((f&CSTOPB)? S_2STOP : S_1STOP);
        if (f & PARENB)
                mode |= (S_PAREN | ((f&PARODD)? 0 : S_PAREVEN));

        switch((f>>4) & 0x03)                   /* bits per char */
        {
        case 0:  break;                 /* 5 bpc */
        case 1:  mode|= S_6BPC;         /* 6 bpc */
                 break;
        case 2:  mode|= S_7BPC;         /* 7 bpc */
                 break;
        case 3:  mode|= S_8BPC;         /* 8 bpc */
                 break;
        }

        /*
         * initialize the usart
         */
        port = (int) &DBbase->USART[unit].cntrl;                        /* usart port */
        picport = (int) &DBbase->PIC[0].msr;   /* pic port for OCW1 */
        s = SPL();
        /* disable PIC interrupts */
        mask = inb(picport)|(3 << (unit * 2));
        outb(picport, mask);                                    /* RxRDY, TxRDY off */
        outb(port, S_IR);                                       /* software reset */
        outb(port, mode);                                       /* serial mode cmd to usart */
        /*
         * turn usart (dtr) on
         * do not enable receiver if not CREAD
         */
        outb(port, S_RTS|S_ER|S_DTR|S_TXEN|
                        ((tp->t_cflag & CREAD)? S_RXEN : 0));

        /* now enable PIC interrupts */
        mask &= (~(3 << unit * 2));
        outb(picport, mask);
        splx(s);

#ifdef DEBUG
        if(i534debug == 0 || i534debug == 2)
                printf("P: mode = %x,mask = %x\n",mode,mask);
#endif
}
```

```
/*
 * This procedure programs an 8253 PIT for operation as a baud rate
 * generator.
 *
 * TITLE:          i53tprog
 *
 * CALL:           i53tprog(timer_port,timer_control_port,mode,speed);
 *
 * CALLS:          outb
 *
 */

i53tprog(timer,pcr,mode,speed)
register int speed,timer;
int mode,pcr;

{

        outb(pcr,mode);                 /* prog mode */
        outb(timer, speed);             /* prog speed */
        outb(timer, (speed>>8));
}


/*
 * This procedure opens one of the 4 lines on the isbc534 board for
 * exclusive use by a user.  The file structure is initialized
 * and control is passed to the tty line discipline read routine,
 * which does the actual open.
 * Not supported is the fifth device which is the parallel port.
 *
 *
 * TITLE:          i534open
 *
 * CALL:           i534open(dev, flag);
 *
 * INTERFACES:     xenix
 *
 * CALLS:          i534param, ttinit, tty open through
                   lineswitch,inb,outb,sleep
 *
 * History: 1/15/82:       Modifed code for multiple i534's to:index a
 *                         configuration table to get the board base address.
 *
 */
```

```
int i534proc();

i534open(dev)
int dev;
{
        struct db534  *DBbase; /* set up the i/o boards base address */
        register struct tty *tp;
        register int unit;

        unit = dev & MINORMSK;                          /* unit <- board num */
        if (unit >= (N534*4)) {                         /* illegal device */
                u.u_error = ENXIO;
                return;
        }
        if (i534alive[unit/4] == 0) {                   /* Board not there! */
                u.u_error = ENXIO;
                return;
        }
        DBbase = (struct db534 *)i534cfg[unit>>2].c_base;
        tp = (struct tty *) &i534tty[unit];
#ifdef DEBUG
        if(i534debug == 0 || i534debug == 2)
                printf("i534open unit = %d,state = %x\n ",unit,tp->t_state);
#endif
        i534addr[unit] = unit;                          /* board and unit number */
        tp->t_proc = i534proc;
        unit %= 4;                                      /* unit <- port num */
        if ((tp->t_state & ISOPEN) == 0 ) {
                /* l001: next few lines conform to new tty stuff */
                ttinit(tp);
                tp->t_oflag = OPOST|ONLCR;                      /* output modes */
                tp->t_iflag = ICRNL|ISTRIP|IXON;                /* input modes */
                tp->t_lflag = ECHO|ICANON|ISIG;
                tp->t_cflag = B9600|CS8|CREAD;

                i534param(dev);                         /* load parameters */

                if(dev & MODEMMSK) {
                        while((inb(&DBbase->USART[unit].cntrl) & DTRON) == 0)
                            sleep((caddr_t)&i534wakeup,TTIPRI);
                        outb(&DBbase->PIC[1].msr ,((inb(&DBbase->
                          PIC[1].msr)) &(~(0x10<< unit)) &TIMERGO));
                                                        /*unmask carrier/detect */
                }
        }
        if ((tp->t_lflag & XCLUDE) && (u.u_uid != 0)) {
                u.u_error = EBUSY;
                return;
        }
        tp->t_state |= CARR_ON;
        (*linesw[tp->t_line].l_open)(tp);
```

```
#ifdef DEBUG
            if (i534debug = = 0 || i534debug = = 2)
                  printf("O:usart %d status = %x\n",unit,
                        inb(&DBbase->USART[unit].cntrl));
#endif
}


/*
 * This procedure performs the close operation on one of the devices of the
 * isbc534. A close masks the device on board; reinstalls the flags that
 * state the device is closed; calls ttyclose to do the operation.
 * Not implemented yet is device 4 which is the parallel port; it is
 * unknown device at this minute.
 *
 * TITLE:           i534close
 *
 * CALL:            i534close(dev, flag);
 *
 * INTERFACES:      xenix
 *
 * CALLS:           tty close thru line discipline
 *
 * History:
 *
 */


i534close(dev)
int  dev;
{
      struct  db534  *DBbase;  /* set up the i/o boards base address */
      register struct tty *tp;
      register int unit;
      int mask;
      int         s;

      unit = dev & MINORMSK;
#ifdef DEBUG
      if(i534debug = = 0 || i534debug = = 2)
                  printf("i534close unit = %d, ",unit);
#endif
      tp = (struct tty *) &i534tty[unit];
      DBbase =  (struct db534 *) i534cfg[unit > > 2].c_base;
```

```
        if (unit < N534*4) {
                unit % = 4;
                if(tp->t_cflag & HUPCL) {
                        /* turn off dtr
                         * carrier will be turned off when intr
                         * from dsr comes in
                         */
                        while ((inb(&DBbase->USART[unit].cntrl) &
                                S_TXRDY) = = 0)
                                ;                               /* wait for txrdy */
                        outb(&DBbase->USART[unit].cntrl, S_ER);/* dtr off */
                }
                (*linesw[tp->t_line].l_close)(tp);
                s = SPL();
                mask = inb(&DBbase->PIC[0].msr) | (3 < < (unit * 2));
                outb(&DBbase->PIC[0].msr, mask);        /* RxRDY, TxRDY off */
                splx(s);
        }
        i534addr[dev&MINORMSK] = (short) 0;
}


/*
 * This procedure interfaces the read request with the system read operation
 * to obtain a byte from the usart. The usart's character is read after an
 * interrupt so this procedure calls the system to wait for the interrupt
 * procedure to pass the character on to the input character queue.
 *
 * TITLE:           i534read
 *
 * CALL:            i534read(dev)
 *
 * INTERFACES:      xenix
 *
 * CALLS:           ttread
 *
 * History:
 *
 */

i534read(dev)
int  dev;
{
        register struct tty *tp;
        register int unit;

        unit = dev & MINORMSK;
#ifdef DEBUG
        if(i534debug = = 0 || i534debug = = 2)
                printf("i534read unit = %d, ",unit%4);
#endif
        tp = (struct tty *) &i534tty[unit];
        (*linesw[tp->t_line].l_read)(tp);
}
```

```
/*
* This procedure is the complement of the i534read routine. A call is
* made to the line discipline write routine, which watches the output
* queue for characters and passes
* the characters from the output queue to the device.
*
* TITLE:            i534write
*
* CALL:             i534write(dev);
*
* INTERFACES:       xenix
*
* CALLS:            ttwrite
*
* History:
*
*/

i534write(dev)
int  dev;
{
        register struct tty *tp;
        register int unit;

        unit = dev & MINORMSK;
#ifdef DEBUG
        if(i534debug = = 0 || i534debug = = 2)
                printf("i534write unit = %d, ",unit%4);
#endif
        tp = (struct tty *) &i534tty[unit];
        (*linesw[tp->t_line].l_write)(tp);
}


/*
* This procedure is called by xenix with interrupts off (SPL) when the
* isbc534 interrupts. The interrupt process polls the 8259's on the isbc534
* to find out which device interrupted. If the device is a usart receiving
* it gets the character, then sends the character to the tty line
* discipline input routine, or restarts output by
* calling i534start depending on which interrupt was set off. The carrier
* detect, ring indicator, present next digit and pit interrupt signals are
* not implemented yet. The present next digit signal comes from the
* external source on line 4.
*
```

```
 * NOTE      : all carrier detect signals both interrupt and latch on the 8255
*             ppi. Refer to the H/W manual for possible uses of these signals
*             (ie ACU | printer applications).
*             The rxrdy/txrdy lines from the older usarts (8251A/s2657 & older)
*             cause giltches on the pic interrupt lines. This is a problem with
*             the usart. If possible replace usart with a newer version.
*
*
* TITLE:            i534intr
*
* CALL:             i534intr(level);
*
* INTERFACES:       xenix
*
* CALLS:            l.d. input routine, i534start
*
* History:          1/13/82:    Condensed the usart Rxrdy/txrdy intr switch to
*                               run more efficiently using an if.. ; Added the
*                               unset of busy flag which gets set in i534start.
*                   1/15/82:    changed variable type to level which was incorrect.
*                               added multiple isbc534 support.
*
*/
int         wakeup();

i534intr(level)
int         level;

{
            struct  db534 *DBbase;   /* set up the i/o boards base address */
            register struct tty *tp;
            char   c;
            register int status;
            int    mask;                    /* mask from PIC */
            int    gotone,board;
            short taddr;

#ifdef DEBUG
            if(i534debug > = 1)
                  printf("i534intr, ");
#endif
```

```
        do {
        gotone = 0;
        for(board = 0;board<N534;board + +) {
            if(i534alive[board]) {
                DBbase = (struct db534 *) i534cfg[board].c_base;
                outb(&DBbase->PIC[0].csr, GETINT);
                status = inb(&DBbase->PIC[0].csr);
                if ((status & GOODINT) = = GOODINT){/* check bit 7 for intr */
                    gotone + +;
                    outb(&DBbase->PIC[0].csr, PIC_EOI);
                    status & = 0x07;                    /* status <- port num/rx tx */
#ifdef DEBUG
if(i534debug> = 1)
        printf("Istatus = %x, ",status);
#endif
                    tp = (struct tty *) (&i534tty[board*4] + (status > > 1));
                    if ((status & 0x01) = = 0){  /* Rxrdy intr */
                        c = inb(&DBbase->USART[status > > 1].data);

                        /* check for error */
                        status = inb(&DBbase->USART[status > > 1].cntrl);
#ifdef DEBUG
if(i534debug> = 1)
        printf("Ic = %c status = %x, ",c,status);
#endif
                        if (status & S_PERROR)
                                c|= PERROR;
                        if (status & S_FRERROR)
                                c|= FRERROR;
                        if (status & S_OVERRUN)
                                c|= OVERRUN;
                        (*linesw[tp->t_line].l_input)(tp,c,0);
                    }else  {                                    /* Txrdy intr */
#ifdef DEBUG
        if(i534debug> = 1)
                printf("Itxrdy, ");
#endif
                        tp->t_state & = ~BUSY;                  /* the character is out */
                        /* the next call should really go thru i534proc() */
                        i534start(tp);                          /* do the next one */
                        if((tp->t_state & OASLP) && (tp->t_outq.c_cc < =
                                        ttlowat[tp->t_cflag & CBAUD])) {
                            tp->t_state & = ~OASLP;
                            wakeup((caddr_t)&tp->t_outq);
                        }
                    }
                }
```

```
            outb(&DBbase->PIC[1].csr, GETINT);
            status = inb(&DBbase->PIC[1].csr);
            if ((status & GOODINT) = = GOODINT) {
            /* check bit 7 for intr */
                gotone + + ;
                outb(&DBbase->PIC[1].csr, PIC__EOI);
                status &= 0x07;                          /* mask off garbage bits */
                if (status > = 4)                        /* carrier detect */
                    tp = (struct tty *)(&i534tty[board*4] + (status-4));

                switch(status)                  {/* switch on interrupt */
                case 0 :                        /* pit 1 cntr 4 */
                    break;
                case 1 :                        /* pit 1 cntr 5 */
                    break;
                case 2 :                        /* ring ind all */
                    wakeup((caddr__t)&i534wakeup);
                    break;
                case 3 :                        /* present next */
                    break;
                case 4 :                        /* port 0 detect*/
                case 5 :                        /* port 1 detect*/
                case 6 :                        /* port 2 detect*/
                case 7 :                        /* port 3 detect*/
                    if((tp->t__state & (CARR__ON|ISOPEN))
                            = = (CARR__ON|ISOPEN)) {
                        signal(tp->t__pgrp, SIGHUP);
                        tp->t__state &= ~CARR__ON;
                        /*
                         * flick dtr off to cause hardware
                         * hang up on modem
                         */
                        taddr = i534addr[tp-i534tty];
                        mask = inb(&DBbase->PIC[1].msr) | (1<<status);
                        outb(&DBbase->PIC[1].msr, mask);
                            /* carrier detect off */
                        while ((inb(&DBbase->USART[taddr&03].cntrl) &
                            S__TXRDY) = = 0)
                            ;                                   /* wait for txrdy */
                        outb(&DBbase->USART[taddr&03].cntrl, S__ER);
                    }
                    break;

                } /* end switch */
            }     /* end if goodint */
        }    /* end if alive */
    }     /* end for */
    } while(gotone);
}
```

```
/*
* This procedure handles I/O functions.  It is called at both
* task time by the line discipline routines, and at interrupt time
* by i534intr(). (NOTE: at this time, i534intr() calls i534start()
* directly.  It does not pass through this routine, since this routine
* does nothing to start output before calling i534start().  This is
* done this way in the interest of efficiency.
* i534proc handles any device dependent functions required
* upon suspending, resuming, blocking, or unblocking output; flushing
* the input or output queues; timing out; sending break characters,
* or starting output.
*
* TITLE:            i534proc
*
* CALL:             i534proc(tp,cmd)
*
* INTERFACES:       xenix (line discipline routines), i534intr
*
* CALLS:            i534start
*
* change history:
*           llk     l001
*                   Added this routine.
*
*/
int i534brk();

i534proc(tp,cmd)
register struct tty *tp;
int             cmd;
{
                register int port;
                struct db534 *DBbase;           /* i/o board base addr */
                short taddr;

#ifdef DEBUG
                if(i534debug >= 1);
                        printf("i534proc cmd = %d",cmd);
#endif
                taddr = i534addr[tp-i534tty];
                switch(cmd) {
                case T_RFLUSH:                  /* flush input queue */
                        return;
                case T_WFLUSH:                  /* flush output queue */
                case T_RESUME:                  /* resume output */
                        tp->t_state &= ~TTSTOP;
                        i534start(tp);          /* start output */
                        return;
                case T_SUSPEND:                 /* suspend output */
                        tp->t_state |= TTSTOP;
                        tp->t_state &= ~BUSY; /* output no longer in progress */
                        return;
```

```
        case T_BLOCK:                                   /* send stop char */
                if (putc(CSTOP, &tp->t_outq) == 0)              {
                        tp->t_state |= TBLOCK;
                        i534start(tp);
                }
                return;
        case T_UNBLOCK:                                 /* send start char */
                if (putc(CSTART, &tp->t_outq) == 0)             {
                        tp->t_state &= ~TBLOCK;
                        i534start(tp);
                }
                return;
        case T_TIME:                                    /* time out */
                tp->t_state &= ~TIMEOUT;
                i534start(tp);
                return;
        case T_BREAK:                                   /* send null for .25 sec */
                DBbase = (struct db534 *) i534cfg[taddr >> 2].c_base;
                port = (int) &DBbase->USART[taddr & 03].cntrl;

                while ((inb(port) & S_TXRDY) == 0)
                        ;                               /* wait for txrdy */
                /* disable receiver, send break */
                outb(port, S_SBRK|S_TXEN);
                timeout(i534brk, tp, HZ/4);
                sleep((caddr_t)&tp->t_state);
                return;
        case T_OUTPUT:                                  /* start output */
                i534start(tp);
        };              /* end switch */

}


/*
 * This procedure starts output on a usart if needed. i534start gets a
 * character from the character queue, outputs the character to the usart,
 * and sets the BUSY flag. The busy flag gets unset when the character
 * has been transmitted by i534intr().
 *
 * TITLE:               i534start
 *
 * CALL:                i534start(tp)
 *
 * INTERFACES:          i534proc
 *
 * CALLS:               timeout, wakeup, getc, outb, SPL
 *
```

```
 * History:          1/13/82: Removed the hardware probing for txrdy and added
 *                            a set of the busy flag which gets unset on txrdy
 *                            interrupt.
 *              l000  plb    6/28/83
 *                    removed race condition
 *              l001  llk    1/29/84
 *                    sys 3 port
 *
 */
int ttrstrt();

i534start(tp)
register struct tty *tp;
{
        register int c;
        int s;
        int cntrlport, dataport;                /* control and data ports for usart */
        struct db534 *DBbase;                   /* i/o board base addr */
        short taddr;

        taddr = i534addr[tp-i534tty];
        DBbase = (struct db534 *) i534cfg[taddr >> 2].c_base;
        cntrlport = (int) &DBbase->USART[taddr & 03].cntrl;
        dataport = (int) &DBbase->USART[taddr & 03].data;

#ifdef DEBUG
        if(i534debug >= 1) {
                printf("\ni534start: unit = %x", taddr);
                printf("ttstate = %x",tp->t_state);
                printf("ustatus = %x\", inb(cntrlport));
        }
#endif
        s = SPL();
        if (tp->t_state&(TIMEOUT|BUSY|TTSTOP)) { /* l001: added TTSTOP */
                splx(s);
                return;
        }
        tp->t_state |= BUSY;        /* l000 */
        splx(s);
        if ((c = getc(&tp->t_outq)) >= 0) {
                if ((tp->t_oflag & OPOST) == 0) {
                        while ((inb(cntrlport) & S_TXRDY) == 0)
                                ;                       /* wait for txrdy */
                        outb(dataport, c);
                }
```

```
                else {                  /* cooked I001 */
                        if (c = = 0200) {
                                if((c = getc(&tp->t_outq))<0) return;
                                if(c>0200){
                                        tp->t_state |= TIMEOUT;
                                        tp->t_state &= ~BUSY;
                                        timeout(ttrstrt, (caddr_t)tp, (c&0177));
                                        return;
                                }
                        }
                        while ((inb(cntrlport) & S_TXRDY) = = 0)
                                                        /* wait for txrdy */
                        outb(dataport, c);
                }   /* else cooked mode */
        }
        else
                tp->t_state &= ~BUSY;    /* I000 */

        if(tp->t_state&TTIOW && tp->t_outq.c_cc = = 0) {
                tp->t_state &= ~TTIOW;
                wakeup((caddr_t)&tp->t_oflag);
        }
        if(tp->t_state&OASLP&&tp->t_outq.c_cc< = ttlowat[tp->t_cflag&CBAUD]) {
                tp->t_state &= ~OASLP;
                wakeup((caddr_t)&tp->t_outq);
        }
}


/*
 * This procedure releases the transmitter output.
 * It is used by the TCSBRK ioctl command.  After .25 sec
 * timeout (see case BREAK in i534proc), this procedure is called.
 *
 * TITLE:           i534brk
 *
 * CALL:            i534brk(addr)
 *
 * INTERFACES:      timeout (through i534proc)
 *
 * CALLS:           wakeup
 *
 * change history:
 *          llk     I001
 *                  Added this routine.
 *
 */
```

```
i534brk(tp)
register struct tty *tp;
{
        register int port;
        struct db534 *DBbase;           /* i/o board base addr */
        short taddr;

        taddr = i534addr[tp-i534tty];
        DBbase = (struct db534 *) i534cfg[taddr >> 2].c_base;
        port = (int) &DBbase->USART[taddr & 03].cntrl;

        /* enable receiver, if supposed to */
        outb(port,S_RTS|S_ER|S_DTR|S_TXEN|((tp->t_cflag&CREAD) ? S_RXEN:0));
        wakeup((caddr_t)&tp->t_state);
}



/*
 * This procedure handles the ioctl system calls for such things as baud
 * rate changes and various hardware control changes from the initial set
 * up.  Currently only baud rate changes are supported.
 *
 * TITLE:          i534ioctl
 *
 * CALL:           i534ioctl(dev, cmd, arg, flag)
 *
 * INTERFACES:     ioctl
 *
 * CALLS:          i534param, ttiocom
 *
 * History:
 *          I001   added new ioctl commands
 *
 */

i534ioctl(dev, cmd, arg, flag)
int     dev;
int         cmd, flag;
faddr_t arg;
{
        register struct tty *tp;
        register int unit;

        unit = dev & MINORMSK;
        tp = (struct tty *) &i534tty[unit];
        if (ttiocom(tp,cmd, arg, dev)) {
                i534param(dev);         /* do it */
        }
}
```

intel®

CHAPTER 5
BLOCK DRIVERS

This chapter describes the elements of XENIX 286 block device drivers. A block device is organized as an array of blocks, each block containing **BSIZE** bytes. The blocks must be randomly accessible in a reasonable time; for example, the kernel might reference block 3, then block 789, and then block 50, and each access should take only a fraction of a second even though the blocks are scattered on the device.

Magnetic disk drives, bubble memories, and RAM disks all qualify as randomly accessible block devices. A RAM disk is simply an area of RAM semiconductor memory that is set aside to simulate a disk drive. The advantage of a RAM disk is that it can be accessed much more rapidly than a magnetic disk; however, the contents of a RAM disk are lost if system power is lost. Some of the advantages of a RAM disk can also be achieved by simply increasing the number of block buffers in the XENIX system. Tape drives do not qualify as block devices, because data on tapes must be accessed sequentially, and simulating random access in software would result in unacceptably poor performance.

A block device may contain one or more XENIX *file systems*. A file system is a hierarchy of directories and files starting at the *superblock* on the device. All manipulation of file system structures is done by kernel code; a block device driver simply reads and writes physical blocks without knowledge of the structure of the file system. The kernel also manages the allocation and deallocation of free space on the device; the block device driver simply reads and writes physical blocks as requested by the kernel.

This chapter begins by describing how blocks are buffered by the kernel and the driver, and then gives a more detailed overview of block drivers. The remainder of the chapter describes a hypothetical hard disk driver. In the hypothetical driver and in actual hard disk drivers, each physical disk is divided into a number of *partitions*. Each partition has a distinct minor device number and a distinct device special file and is a distinct logical device. Each partition can contain a XENIX file system and be opened, closed, and accessed independent of the other partitions. Thus when discussing block devices, *device* can mean either a logical device (such as a disk partition) or an entire physical device that may contain several partitions.

Block Buffering

This section describes how the kernel and block device drivers buffer blocks to be transferred between a block device and user memory. The functionality described is provided almost entirely by the kernel and you can write a driver without understanding most of the material in this section. However, this section will help you understand the **buf** and **iobuf** data structures used by block drivers and how the kernel attempts to minimize and optimize disk accesses.

The kernel maintains a global pool of block buffers, each **BSIZE** bytes, that it uses as needed. Each buffer is referenced by a *buffer header* that contains information about the buffer and what it is used for. The buffer header is defined by the data type **struct buf** in the include file **buf.h**, which is listed in Appendix E of this manual.

Each block device capable of operating concurrently has a separate device-specific header of type **struct iobuf**. This header references the buffers being used for the device. The device-specific header is defined in the include file **iobuf.h**, which is listed in Appendix F of this manual.

There are three lists that a block buffer can be on; a buffer is always on one of these lists and may be on two simultaneously:

1. The kernel's *free list* contains all block buffers available for allocation or reuse. The free list is circular and doubly linked. The **av_forw** pointer in the buffer header points to the next buffer header on the free list. The **av_back** pointer in the buffer header points to the previous buffer header on the free list. The buffer is on the free list whenever the **B_BUSY** flag in the **b_flags** word of the buffer header is clear (0). The kernel handles the free list; the driver never needs to manipulate it. There is only one systemwide free list and all available block buffers are on it.

2. The driver's *active list* contains all block buffers for which the driver has been called to perform a read or a write, but for which I/O has not been completed. The head of the active list is the device-specific header of type **struct iobuf**. The active list is circular and doubly linked. The **av_forw** pointer in the buffer header points to the next buffer header on the active list. The **av_back** pointer in the buffer header points to the previous buffer header on the active list. The **b_actf** and **b_actl** pointers in the device-specific header reference the first and last buffer headers in the active list respectively. If the active list is empty, then **b_actf** and **b_actl** should each contain the address of the device-specific header (i.e., the header points to itself if the list is empty). The buffer is on the active list whenever the **B_BUSY** flag in the **b_flags** word of the buffer header is set (1). The driver handles the active list, calling the kernel **disksort** routine to insert a buffer in the list. Driver code must initialize the active list and remove buffers when I/O is completed. Note that there are multiple active lists, one for each distinct 16-bit block device number.

3. The device-specific *device list* contains all block buffers that contain current valid copies of blocks on the associated device. The device lists act as a cache, so that disk blocks mirrored in the device lists need not be read from disk. Also, disk blocks being written can be put on both the device list and the free list but not actually transferred until a shortage of blocks causes the written blocks to be needed elsewhere. The head of a device list is the device-specific header of type **struct iobuf**. Each device list is circular and doubly linked. The **b_forw** pointer in the buffer header points to the next buffer header on the device list. The **b_back** pointer in the buffer header points to the previous buffer header on the device list. The **b_forw** and **b_back** pointers in the device-specific header reference the first and last buffer headers in the device list respectively. If the device list is empty, then **b_forw** and **b_back** should each contain the address of the header (i.e., the header points to itself if the list is empty). A buffer is on a device list whenever the **B_DONE** flag in the **b_flags** word of the buffer header is set (1). The kernel handles the device lists; driver code does not need to manipulate these lists. Note that there are multiple device lists, one for each distinct 16-bit block device number.

   The purpose of the device lists is to allow the kernel to access frequently referenced information on block devices without having to actually access the device. If the kernel needs to read block *b* on device *d*, it first searches the device list for the device; if block *b* is found, the kernel does not issue a read request to the device driver. The kernel uses an auxiliary hash table to reduce the time required for searches. The hashing and searching are done entirely by the kernel and do not affect the driver.

A block buffer is usually on the free list and a device list at the same time. Block buffers on the free list are ordered using the Least Recently Used algorithm. When a block that contains valid device data is read, it is put at the end of the free list. Block buffers to be allocated are removed from the front of the free list and will either be block buffers that do not contain valid device data (are not on a device list) or those block buffers with data that has not been referenced for the longest time. Because of this algorithm, frequently read blocks, such as those that contain directory information, are normally in main memory, reducing disk accesses.

A block buffer can also be on an active list and a device list at the same time. When a block is being written, it is placed on the device list and then goes on the active list until it is successfully written. This is understandable because the block buffer contains valid data even before the write operation completes.

A block buffer is on the free list and no other list if it contains no cached data and is not being used for a read or a write. A block buffer is on an active list and no other list if it is being used to read in a block of data. A block buffer is never on a device list and no other list; a block buffer on a device list is always also on either the free list or an active list. Finally, a block buffer is never on the free list and an active list at the same time.

When a block is written, the kernel must also check the cache formed by the collected device lists, to invalidate any previous copy of that block that is in the cache. This ensures that the cache contains only the most recent version of any block.

The buffer header data structure, of type **struct buf**, contains the following fields:

b_flags flag word containing the following flags:

B_READ set (1) if block is to be read, clear (0) if block is to be written.

B_DONE set (1) if block is on device list (in cache).

B_ERROR set (1) by driver if transfer failed.

B_BUSY set (1) by kernel if the driver has been called with the buffer; the driver must call **disksort** to insert the buffer in the active list; the driver must handle I/O transfers and then unlink the buffer from the active list; the driver must then call the kernel routine **iodone** for the buffer; **iodone** clears **B_BUSY** and places the buffer on the free list and on the device list unless an error occurred.

There are other flags used only by the kernel that do not need to be understood to write a driver. All the flag names are defined as integer constants that are the bit masks used to test, set, or clear the actual flags. For example **bp->b_flags&B_READ** can be used to test the read flag; **bp->b_flags |= B_READ** can be used to set the read flag; **bp->b_flags &= ~B_READ** can be used to clear the read flag. The include file **buf.h** also defines the value 0 as the so-called "pseudo-flag" **B_WRITE**. Be careful when using **B_WRITE** in programs, as it is not a bit mask and cannot be used to test, set, or clear a flag in the same way as the other constants.

b_forw the forward pointer for the device list (cache), used if **B_DONE** is set.

b_back the backward pointer for the device list (cache), used if **B_DONE** is set.

av_forw the forward pointer for the free list (available list) if **B_BUSY** is clear, else the forward pointer for the active list (pending transfers) if **B_BUSY** is set.

av_back the backward pointer for the free list (available list) if **B_BUSY** is clear, else the backward pointer for the active list (pending transfers) if **B_BUSY** is set.

b_dev the device number, major and minor.

b_bcount number of bytes to be transferred, always an exact multiple of **BSIZE** for block devices.

b_paddr physical address in main memory of the buffer associated with this header.

b_blkno block number on device.

b_error error code to be returned in **u.u_error** if **iodone** is called on the block with **B_ERROR** set.

b_resid number of bytes not transferred when an error was encountered.

b_cylin cylinder number, computed by driver and used by **disksort** routine to order requests in the active list.

The device-specific header, of type **struct iobuf**, contains the following fields:

b_flags NOT USED.

b_forw the head forward pointer for the device list (cache) for this device.

b_back the head backward pointer for the device list (cache) for this device.

b_actf the head forward pointer for the active list (pending transfers).

b_actl the head backward pointer for the active list (pending transfers).

b_dev device number.

The remaining fields of **iobuf**, not referenced by the kernel at all, are really private to the device driver and may be used or ignored as desired by the driver writer:

b_active busy flag.

b_errcnt error count (for soft error retry and recovery).

io_addr device register address for memory-mapped device registers.

io_s1, io_s2
"space for drivers to leave things." The driver writer should define appropriate driver variables as needed and not use these fields. They are defined only for compatibility with existing drivers.

Block Driver Overview

This section provides an overview of a block driver, based on an understanding of the kernel's buffering system and how it works.

A block device driver normally supports both a block interface to the device and a character interface to the device. The character interface is also known as the "raw" interface. The character interface is invoked by opening a device special file that specifies that it is a character device with the major number of the block device driver. Because a character device has been opened, the kernel calls the driver via the switch table **cdevsw**, calling the routines **ixxxopen, ixxxclose, ixxxread, ixxxwrite,** and **ixxxioctl** as needed. The character interface is provided for two reasons. First, reading and writing via the character interface bypasses the block buffering system, which is more efficient for some applications, such as copying an entire disk. Second, the character interface provides the **ixxxioctl** routine, used for disk formatting. Because this routine is provided in the character interface, it is not part of the block interface. Though called the "character" interface, the buffers used for reading and writing must have a size equal to an integer multiple of the system buffer size **BSIZE.**

The block interface to a device is normally invoked by accessing a file or directory in a file system on the device. A kernel variable **root** specifies the device number to use for the root directory, which is the beginning of a chain that leads to any mounted file system and block device. The major device number for the device containing the desired file or directory is used to index the block device switch table **bdevsw,** which for each block major number contains the routines **ixxxopen, ixxxclose,** and **ixxxstrategy,** and also the data structure **ixxxtab,** the device-specific header for lists of buffers. The header **ixxxtab** is of type **struct iobuf.** Accessing a file or directory always uses the block interface to a device, never the character interface.

The block interface to a device can also be invoked by opening the device special file for the device in the directory **/dev.** The device special file specifies that the device is a block device and specifies major and minor device numbers. The minor number can distinguish multiple drives, partitions in a partitioned device, or type of media if different types are supported. Note that the character and block interfaces to a device have separate device special files with distinct names but with the same major and minor numbers. To open a device special file directly, a user or program must normally have super-user privileges if it previously contained a file system, or else be the owner of the special file. When either device special file for a block device is opened directly, the resulting file descriptor corresponds to a file that contains all the bytes of the corresponding disk or partition. Reading or writing this file reads or overwrites bytes of the disk or partition, without regard to directory structure, allocated or free blocks, or file allocations. Writing to this "file" can corrupt or destroy information in the target partition or disk.

A block device driver contains the following routines and data structure referenced by the kernel, for both the block and character interfaces:

ixxxinit() — Called at system startup to determine which devices managed by the driver are present and to initialize the devices and associated data structures.

ixxxopen(dev, oflag) — Called when the device is mounted. (The root device is mounted by the kernel at system startup, resulting in an **ixxxopen** call.) Also called if a device special file for the device is opened directly.

ixxxclose(dev, oflag) — Called when the device is unmounted. (The kernel does not allow the root device to be unmounted.) Also called when a device special file for the device that was opened directly is closed.

ixxxstrategy(bp) — Called with a buffer header containing a read or write request for a device managed by the driver. Inserts the request into the device's active list sorted by cylinder number. Ensures that I/O is ongoing or started.

ixxxintr(level) — Called when a device managed by the driver interrupts the CPU. Determines which device sent the interrupt. Acknowledges the interrupt and reads device status to check for errors. Calls **iodone** to return the associated buffer to the kernel. Starts the next I/O request, if any.

ixxxread(dev) — Character interface routine. Called to transfer data directly from the device to user memory. Calls the kernel routine **physio**, which calls **ixxxstrategy**.

ixxxwrite(dev) — Character interface routine. Called to transfer data directly from user memory to the device. Calls the kernel routine **physio**, which calls **ixxxstrategy**.

ixxxioctl(dev, cmd, cmdarg, oflag) — Character interface routine. Called for special device functions, such as formatting a disk.

struct iobuf ixxxtab — Device-specific header for active list and device list of buffer headers.

Driver Files

The driver code is contained in three files:

- **sys/h/ixxx.h** defines constants used by the driver.
- **sys/cfg/cxxx.c** defines data structures used by the driver.
- **sys/io/ixxx.c** defines the driver routines.

The file **sys/h/ixxx.h** is included by the other two files:

```
#include "../h/ixxx.h"
```

The main driver file **sys/io/ixxx.c** should also include these files:

```
#include "../h/buf.h"          /* for buf data structure  */
#include "../h/iobuf.h         /* for iobuf data structure */
#include "../h/param.h"        /* for BSIZE - buffer size    */
#include "../h/user.h"         /* for u structure and error codes */
```

Adding the device to the configuration also requires editing the files **sys/conf/master** and **sys/conf/xenixconf,** as described in Chapter 6, "Adding Drivers to the Configuration."

The following sections describe these files in more detail and include some example code for a hypothetical hard disk driver.

Driver Constants

These constants are defined in the example **sys/h/ixxx.h:**

```
#define ixxx_NUM    4       /* number of drives supported by driver   */
#define ixxx_NP     8       /* max # of partitions per drive          */

#define ixxx_CPD   600      /* cylinders per drive                    */
#define ixxx_TPC     4      /* tracks per cylinder                    */
#define ixxx_SPT    10      /* sectors per track                      */
#define ixxx_bPS   512      /* bytes per sector                       */
#define ixxx_SPC   (ixxx_SPT*ixxx_TPC)
                            /* sectors per cylinder                   */
#define ixxx_BPC   ((ixxx_SPC * ixxx_bPS)/BSIZE)
                            /* blocks per cylinder (calculation
                               must not overflow and must have no
                               remainder)                             */
/* There must be an integer number of sectors per block and an integer
   number of blocks per cylinder.                                      */
```

```
/* register offsets */
#define RCMD      0       /* offset of  8-bit command register          */
#define RSTAT     1       /* offset of  8-bit status  register          */
#define RCYL      2       /* offset of 16-bit cylinder register         */
#define RTRK      4       /* offset of  8-bit track register            */
#define RSEC      5       /* offset of  8-bit sector register           */
#define RADRL     6       /* offset of 16-bit register containing
                             low 16-bits of transfer address in
                             physical memory                            */
#define RADRH     8       /* offset of 8-bit register containing
                             high 8-bits of transfer address in
                             physical memory                            */
#define RSCNT     9       /* 8-bit number of sectors to transfer        */

/* command codes */
#define CRESET    0       /* Reset device and controller.               */
#define CENINT    1       /* Enable device interrrupts.                 */
#define CDISINT   2       /* Disable device interrupts.                 */
#define CREAD     3       /* Read contiguous sectors from device.       */
#define CWRITE    4       /* Write contiguous sectors to device.        */
#define CFORMAT   5       /* Format the media in the device.            */

/* status register bit masks */
#define SINT      1       /* This drive interrupted.                    */
#define SERR      2       /* An error in the last operation.            */
#define SBUSY     4       /* The drive is busy and cannot accept
                             a command (except CRESET, which is
                             always accepted).                          */
#define SENA      8       /* Drive interrupts are enabled.              */

/* device minor number format is xxxddppp, xxx = not used, should be 0
   dd = drive number, ppp = partition number */
#define drive(dev)    ((dev&037) >> 3)
#define part(dev)     (dev & 07)

/* ioctl command code for formatting a drive */
#define IOC_FMT   0

/* status values used in device status array */
#define ABSENT    0       /* Drive is not in the system.                */
#define PRESENT   1       /* Drive is present but not open.             */
#define OPEN      2       /* Drive is present and open.                 */
#define LOCKED    3       /* Drive should not be accessed (e.g.,
                             it is being formatted).                    */
/* ABSENT applies to an entire drive and only need to be
   assigned in the ixxx_sts element for partition 0 to take effect,
   e.g.: ixxx_sts[dr][0] = ABSENT; */
```

Driver Data Structures

These data structures are defined in the example **sys/cfg/cxxx.c:**

```
/* base addresses for controller registers in port address space */
unsigned ixxx_adr[ixxx_NUM] = { 0x8000, 0x8020, 0x8040, 0x8060 };

/* partition sizes for the first hard disk drive, which holds the
   root file system, swap space, and user files.                    */
#define ROOTSZ  200             /* # of cylinders in root partition    */
#define SWAPSZ  100             /* # of cylinders in swap partition    */
#define USERSZ  (ixxx_CPD - (ROOTSZ + SWAPSZ))
                                /* # of cylinders in user partition    */

/* partition sizes for subsequent hard disks; each disk is divided
   into two "extra" partitions.                                     */
#define EX1SZ   300             /* # of cylinders in extra partition 1 */
#define EX2SZ   (ixxx_CPD - EX1SZ)
                                /* # of cylinders in extra partition 2 */

struct partitn {
    unsigned  cyl;              /* starting cylinder of partition      */
    unsigned  len;              /* # of blocks in partition            */
}

/* partition table */
struct partitn ixxx_par[ixxx_NUM][ixxx_NP] = {
    0, ixxx_CPD*ixxx_BPC,       /* entire disk                         */
    0, ROOTSZ*ixxx_BPC,         /* root partition                      */
    ROOTSZ, SWAPSZ*ixxx_BPC,    /* swap partition                      */
    (ROOTSZ+SWAPSZ), USERSZ*ixxx_BPC,/* user partition                 */
    0,0,0,0,0,0,0,0,            /* partitions 4..7 not used            */

    0, ixxx_CPD*ixxx_BPC,       /* entire disk                         */
    0, EX1SZ*ixxx_BPC,          /* extra partition                     */
    EX1SZ, EX2SZ*ixxx_BPC,      /* extra partition                     */
    0,0,0,0,0,0,0,0,0,0,        /* partitions 3..7 not used            */

    0, ixxx_CPD*ixxx_BPC,       /* entire disk                         */
    0, EX1SZ*ixxx_BPC,          /* extra partition                     */
    EX1SZ, EX2SZ*ixxx_BPC,      /* extra partition                     */
    0,0,0,0,0,0,0,0,0,0,        /* partitions 3..7 not used            */

    0, ixxx_CPD*ixxx_BPC,       /* entire disk                         */
    0, EX1SZ*ixxx_BPC,          /* extra partition                     */
    EX1SZ, EX2SZ*ixxx_BPC,      /* extra partition                     */
    0,0,0,0,0,0,0,0,0,0         /* partitions 3..7 not used            */
}

/* device headers for lists of active buffers and cached buffers. */
struct iobuf ixxxtab[ixxx_NUM];
```

```
/* buffer headers for raw transfers */
struct buf ixxx_raw[ixxx_NUM];

/* status for each drive and partition */
unsigned ixxx_sts[ixxx_NUM][ixxx_NP];
```

ixxxinit Procedure

```
ixxxinit();
{
int dr;       /* drive index */
int pa;       /* partition index */

      /* loop through drives */
      for (dr = 0; dr < ixxx_NUM; dr++) {

            /* Check to see if drive is present. */
            out(ixxx_adr[dr] + RCYL, 0x5aa5);
            if (in(ixxx_adr[dr] + RCYL) != 0x5aa5) {
                  /* Drive is not present. */
                  printf("ixxx drive %d ABSENT\n", dr);
                  ixxx_sts[dr][0] = ABSENT;
                  }
            else {
                  /* Drive is present. */
                  printf("ixxx drive %d PRESENT\n", dr);
                  for (pa = 0; pa < ixxx_NP; pa++) {
                        ixxx_sts[dr][pa] = PRESENT;
                  }

                  /* Reset the drive. */
                  outb(ixxx_adr[dr] + RCMD, CRESET);

                  /* Initialize the active list for the drive. */
                  ixxxtab[dr].b_actf = &ixxxtab[dr];
                  ixxxtab[dr].b_actl = &ixxxtab[dr];
            }
      }
}
```

This procedure is called via the switch **dinitsw** during system initialization. **ixxxinit** checks for the presence of each possible device handled by the driver. This checking is done by writing a test pattern to a device register that will store it and then reading the register. If the pattern is read back, the device is present. In the example procedure, the cylinder number register is used for the test. The pattern 0x5aa5 is chosen to include both zero bits and one bits, as an absent device is most likely to be read as all zero bits or all one bits. **ixxxinit** should call **printf** with a message for each device checked, indicating if it is absent or present. (The kernel version of **printf** writes its output directly on the system console device, shutting off interrupts to do so; it should be used sparingly or just for debugging.) For each drive present, **ixxxinit** resets it to place it in a known state and also initializes the drive's **ixxxtab** header.

ixxxopen Procedure

```
ixxxopen(dev, oflag)
dev_t  dev;      /* device number, major/minor */
int    oflag;    /* open mode flags, NOT USED */
{
int dr  =  drive(dev);        /* drive number */
int pa  =  part(dev);         /* partition number */

      if (dr>=ixxx_NUM || ixxx_sts[dr][0]==ABSENT)
            u.u_error = ENXIO;    /* "No such device or address" */

      if (ixxx_sts[dr][0]==LOCKED)
            u.u_error = EBUSY;    /* "Mount device busy" */

      else {
            ixxx_sts[dr][pa] = OPEN;
            outb(ixxx_adr[dr] + RCMD, CENINT);
      }
}
```

ixxxopen is called by the kernel via **bdevsw** when the device **dev** is mounted (either by the **mount** system call or at system initialization for the root, pipe, or swap devices). **ixxxopen** can also be called via **cdevsw** or **bdevsw** if the device special file for **dev** is opened with the **open** system call. Note that **ixxxopen** is *not* called if a file on the device is opened.

ixxxopen first checks its parameters, ensuring that the drive number is valid and that the device is neither absent nor locked. Errors are indicated by assigning an error code to **u.u_error.** If no errors are encountered, **ixxxopen** assigns the device status as **OPEN** and enables device interrupts.

Note that because the kernel always calls **ixxxopen** for a device before calling other driver routines, **ixxxopen** is the only routine that must validate the device number and check for device **ABSENT** status.

For devices that use different types of media (e.g., either single density or double density flexible disks), the media type can be encoded in the minor number, and **ixxxopen** can determine media type and configure the device controller and driver tables accordingly.

ixxxclose Procedure

```
ixxxclose(dev, oflag)
dev_t  dev;       /* device number, major/minor */
int    oflag;     /* open mode flags, NOT USED */
{
int dr = drive(dev);   /* drive number */
int pa = part(dev);    /* partition number */

      if (ixxx_sts[dr][0] = = LOCKED)
            u.u_error = EBUSY;     /* "Mount device busy" */

      else {
            ixxx_sts[dr][pa] = PRESENT;
      }
}
```

ixxxclose is called by the kernel via **bdevsw** when the device **dev** is unmounted (with the **umount** system call) or at system shutdown. **ixxxclose** can also be called via **bdevsw** or **cdevsw** if a device special file for **dev** was opened directly and then closed with the **close** system call. Note that **ixxxclose** is not called if a file on the device is closed.

If the device is locked, **ixxxclose** indicates an error by assigning a code to **u.u_error.** Otherwise, **ixxxclose** simply updates the device status and returns.

ixxxstrategy Procedure

```
ixxxstrategy(bp)
register struct buf  *bp;
{
int dr  =  drive(bp->bdev);    /* drive number */
int pa  =  part(bp->bdev);     /* partition number */
int bl  =  bp->b_blkno  +  (bp->b_bcount + BMASK) > > BSHIFT);
                               /* last block number */
int msk;                       /* for saving interrupt mask */

      if (bp->b_blkno > = ixxx_par[dr][pa].len ||
          (bl > = ixxx_par[dr][pa].len && ~(bp->b_flags & B_READ)) {
           /* Indicate error. */
           bp->b_flags |= B_ERROR;
           bp->b_error = ENXIO;  /* code to assign to u.u_error */
           iodone(bp);
           }

      else {
            if (bl > = ixxx_par[dr][pa].len) {
                  bp->b_resid = bp->b_count - BSIZE*
                              (ixxx_par[dr][pa].len - bp->b_blkno + 1);
                  bp->b_count -= bp->b_resid;
            }

            bp->b_cylin = ixxx_par[dr][pa].cyl + bp->b_blkno/ixxx_BPC;
            msk = splbuf();
            disksort(&ixxxtab[dr], bp);
            if (!ixxxtab[dr].b_active)
                  ixxxstart(dr);
            splx(msk);
      }
}
```

This procedure is called by the kernel via **bdevsw** or from **physio** when a block must be physically read from or written to a device managed by the driver. **ixxxstrategy** first checks block number within partition and last block requested within partition (if a write). A block out of bounds error is handled by setting the **B_ERROR** bit in the **b_flags** field of the buffer header and calling **iodone.**

If the request is a read request and extends beyond the last block of the partition, then the request is truncated to fit.

For valid requests, the cylinder number is computed and stored in the buffer header for use by the **disksort** routine. **splbuf** and **splx** are used to guarantee exclusive access to the active list for calling **disksort** and **ixxxstart. disksort** sorts the request into the active list based on cylinder number, to reduce disk head movement. **ixxxstart** needs to be called to start I/O if there is no I/O active for the drive. When there is an active transfer, the completion interrupt will call **ixxxintr,** which will start the next transfer. Only when there is not an active transfer (not an interrupt yet to happen) for the drive is **ixxxstart** called to "prime the pump" of the interrupt-driven cycle. Note that both **disksort** and **ixxxstart** must be called with interrupts locked out (to at least the level of **splbuf).**

Finally, note that **ixxxstrategy** can be called to transfer multiple blocks, not just single blocks. Examples of multiple block transfers are program loading, process swapping, and some uses of the raw interface.

ixxxstart Procedure

```
ixxx_start(dr)
int dr;          /* validated drive number */
{
register     struct buf *bp = ixxxtab[dr].b_actf;
register     unsigned  sec;
register     unsigned  adr;
register     int       pa;

     if (bp != &ixxxtab[dr]) {
          /* active list not empty */
          pa = part(bp->b_dev);
          sec = (unsigned) bp->b_blkno*(unsigned)(BSIZE/ixxx_bPS);
          adr = ixxx_adr[dr];
          out(adr + RCYL, ixxx_par[dr][pa].cyl + sec/ixxx_SPC);
          sec %= ixxx_SPC;
          outb(adr + RTRK, sec/ixxx_SPT);
          outb(adr + RSEC, sec%ixxx_SPT);
          outb(adr + RSCNT, bp->b_bcount/ixxx_bPS);
          out(adr + RADRL, bp->b_paddr & 0xffff);
          outb(adr + RADRH, bp->b_paddr>>16);
          outb(adr + RCMD, ((bp->b_flags&B_READ)?CREAD:CWRITE);
          ixxxtab[dr].b_active = 1;
          }
     else
          /* active list empty */
          ixxxtab[dr].b_active = 0;
}
```

ixxxstart is called from **ixxxstrategy** (process-time) and from **ixxxintr** (interrupt-time) to start the next disk transfer (if any) for a drive. Interrupts must be locked out when **ixxxstart** is called. **ixxxstart** maintains the flag **b_active** in the **ixxxtab[dr]** header; **b_active** is set (1) if the device controller has been sent a transfer and a completion interrupt can be expected. **b_active** is clear (0) if no transfer is outstanding and no completion interrupt can be expected. If **b_active** is 0, **ixxxstrategy** must call **ixxxstart** to start I/O.

Note that **ixxxstart** does no validation of the requested block number and byte count; these values are validated by **ixxxstrategy.**

ixxxintr Procedure

```
ixxxintr(level)
int level;    /* interrupt level from 8259A Programmable
                 Interrupt Controller (PIC), NOT USED */

{
register int dr;
register int sts;
register struct buf *bp;

      for  (dr = 0; dr < ixxx_NUM; dr++) {
            if (ixxxtab[dr].b_active &&
                ((sts = inb(ixxx_adr[dr] + RSTAT) & SINT)) {
                   bp = ixxxtab[dr].b_actf;
                   if (sts & SERR) {
                        bp->b_flags |= B_ERROR;
                        deverr(&ixxxtab[dr], bp, sts, 0);
                   }

                   ixxxtab[dr].b_actf = bp->av_forw;
                   if (ixxxtab[dr].b_actl == bp)
                        ixxxtab[dr].b_actl = bp->av_back;
                   bp->b_resid = 0;
                   ixxx_start(dr);
                   iodone(bp);
            }
      }
}
```

ixxxintr is called by the kernel for each interrupt that occurs for a device managed by the driver. This example driver presumes that all devices managed by the driver use the same interrupt level, requiring **ixxxintr** to poll for interrupts. The **for** loop is for polling each device. The major **if** statement ignores interrupts unless the device status indicates that the drive interrupted and the **b_active** field of **ixxxtab[dr]** indicates that an interrupt is expected.

If an interrupt is handled, an error indication in the drive status causes the **B_ERROR** flag in the buffer header to be set and **deverr** to be called to report the error. (An actual driver might include more sophisticated error handling, such as retry for soft errors.)

For any transaction, whether it ended with or without error, the buffer header for the transaction is removed from the device's doubly-linked active list. **ixxxstart** is called to start the next transaction in the active list (if any). **iodone** is then called to dispose of the buffer for the transaction just completed. **iodone** handles copying any read data to the calling process's address space, placing the buffer on the device list and the free list, and updating the buffer state flags.

ixxxread and ixxxwrite Procedures

```
ixxxread(dev)
dev_t   dev;        /* device number,      major/minor */
{
        physio(ixxxstrategy, &ixxx_raw[drive(dev)],          dev, B_READ);
}

ixxxwrite(dev)
dev_t   dev;        /* device number,      major/minor */
{
        physio(ixxxstrategy, &ixxx_raw[drive(dev)],          dev, B_WRITE);
}
```

These procedures are called in response to **read** and **write** system calls on a file descriptor that has been opened for "raw" character I/O to the device special file. These procedures are never called when reading or writing ordinary files; file system I/O always uses the block I/O interface.

When a block device special file is opened directly, either in block or character mode, the "file" opened is the sequence of all the physical bytes on the device or in the disk partition. Reading and writing a block device at this level ignores all the structures placed on the disk by the file system: super-blocks, inodes, directories, etc. There are obvious dangers in writing a block device at this level. Normally only the super-user can open a device special file and use these facilities.

Such low-level I/O to a block device is used for copying disks (as byte-for-byte images vs. individually copying files) and may be useful for other system functions, such as backing up devices or troubleshooting block devices. A separate interface is provided to this low-level I/O for efficiency reasons. It is desirable to bypass the kernel's complex (and normally desirable) block buffering algorithms when an entire disk is being copied.

Instead of copying data from disk to kernel buffer and then to user space and then to another kernel buffer and finally to the destination device, with all the buffer manipulation code as well, raw I/O copies one or multiple buffers directly to an area of user memory (**ixxxread**); raw I/O can then be used to copy the blocks directly to the destination device (**ixxxwrite**).

For raw I/O, an area of user memory that is an integer multiple of **BSIZE** bytes in size must be set aside as a buffer for the blocks being read. This area is specified as the source and destination for the **read** and **write** system calls. For raw I/O, the byte count specified to these system calls must be an integer multiple of **BSIZE** bytes.

The **physio** routine called by both **ixxxread** and **ixxxwrite** works as follows:

1. The buffer header that is its second argument is used as a buffer header to "fool" the block driver routines. **physio** assigns fields in this header to reference the area of user memory to be used as the source or destination of the transfer. (The address and count are available to **physio** in the **u** structure.)

2. **physio** then calls the driver's own strategy routine, its first argument, to do the transfer directly to or from user memory. To the strategy routine, the request appears in a buffer header like any other buffer header. Thus the driver calls **physio** for raw I/O and is then called by **physio.** The other two arguments to **physio** are the device number and a flag indicating whether the operation is a read or a write.

3. **physio** also locks the calling process into its present memory location and prevents the calling process from being swapped. This is to ensure that the buffer in user memory is there when the driver does the transfer.

ixxxioctl Procedure

```
ixxxioctl(dev, cmd, cmdarg, oflag);
dev_t dev;          /* device number, major/minor */
int cmd;            /* command code */
int *cmdarg;        /* pointer to arguments in user memory */
int oflag;          /* open flags for the device, NOT USED */
/*
```

ixxxioctl is called in response to an **ioctl** system call for a device. Note that this system call is only available via the raw, character interface to a block device. Typically the only command code defined for a block device is for formatting, e.g., formatting a track of the device based on a table given as the **cmdarg** parameter. The code for such formatting is device-dependent. Handling a formatting request may involve the **ixxxstrategy, ixxxstart,** and **ixxxintr** routines if the request is inserted into the normal stream of requests for the device.

intel®

CHAPTER 6
ADDING DRIVERS TO THE CONFIGURATION

This chapter describes how to add a new device driver to XENIX:

1. Edit a **master** file to add information about your driver.
2. Edit a **xenixconf** file to add information about your driver.
3. Edit two **makefile** files to add information about your driver.
4. Use the **make** command to create a new XENIX kernel.
5. Edit **/dev/makefile** to add **mknod** commands for your device; then execute **make** to create the device special files for your device.
6. Restart your system using the new kernel, with the device hardware installed in your system.

The last section of this chapter describes how to delete a device driver from your XENIX system.

Readers of this chapter should also read the *XENIX 286 Installation and Configuration Guide*, which contains more information about configuration, including device driver configuration.

This chapter assumes that you have written your driver, as described in Chapters 1-5, in three files:

- **sys/h/ixxx.h** defines constants used by the driver.
- **sys/cfg/cxxx.c** defines configuration data structures used by the driver.
- **sys/io/ixxx.c** defines the driver routines and any driver data structures that are independent of system configuration.

The location of the **sys** directory in your system depends on whether or not you have purchased XENIX 286 source code. In systems without source code, **sys** is contained in the root directory and has an absolute path name of **/sys**. In systems with source code, **sys** is contained in the **usr** directory and has an absolute path name of **/usr/sys**.

You must be logged in as the super-user to perform many of the tasks described in this chapter. When you restart the system with the new kernel, there should be no other users on the system.

Editing the master File

You must add one line of information about your device driver to the file **sys/conf/master.** This is a text file and can be changed with any of the editors.

The file begins with comment lines, indicated by asterisks (*) in column 1. The table of devices in the configuration follows the initial comments and is shown in Figure 6-1. A sample **master** listing is in Appendix G of this manual.

There are 14 fields in the line that describes a device, but some fields are unused for particular devices. To fill in the line, you must know the answers to these questions:

1. What is the name of your device? What is the prefix used for your driver routines, if different from the device name?

2. Does your device support a block interface? If it does, then what major number do you want to use for the block interface?

3. Does your device support a character interface? If it does, then what major number do you want to use for the character interface?

4. Does your device use interrupts? If so, what interrupt level(s) does it use?

5. What standard driver routines are not present in your driver and should be replaced by **nodev** or **nulldev** in the **cdevsw** or **bdevsw** tables?

6. What is the maximum number of boards handled by your driver?

```
* The following devices are those that can be specified in the system
* description file.  The name specified must agree with the name shown.
*
* The first twelve entries in both the "bdevsw" and the "cdevsw" are
* reserved for use as block devices. The last four of these entries
* are reserved for additional Intel devices and customer block devices.
* All block devices have the same "bdevsw" and "cdevsw" number.
* The "cmaj" number 1 is reserved for use by the memory driver.
*
*name    vsiz  msk    typ   hndlr  na  bmaj  cmaj  #  na  vec1  vec2  vec3  vec4
*  1       2    3      4      5    6    7     8    9  10   11    12    13    14
*-------------------------------------------------------------------------------
i215       1   0137   014    i215   0    0     0    2  -1  0005    0     0    0a
i216       1   0137   014    i216   0    2     2    2  -1  0005    0     0    0a
i214       1   0137   014    i214   0    3     3    2  -1  0005    0     0    0a
i208       1   0137   014    i208   0    4     4    2  -1  0003    0     0    0a
ramd       0   0136   054    ramd   0    5     5    1  -1     0    0     0    0a
xlog       1   0137   014    xlog   0    6     6    2  -1  0005    0     0    0a
```

Figure 6-1. Device Table from **sys/conf/master**

```
*
* The next twelve entries in the "cdevsw" are reserved for character
* devices. The "cmaj" number 12 is reserved for use by the tty driver.
*
lp      1   0132   004   lp     0   0   13   1 -1   0107   0    0   0a
i74     1   0137   004   i74    0   0   14   1 -1   0006   0    0   0a
i188    1   0137   004   i188   0   0   15   2 -1   0003   0002 0   0a
i552    1   0137   004   i552   0   0   16   1 -1   0004   0    0   0a
i278    1   0137   004   i278   0   0   17   1 -1   0003   0    0   0a
i544    1   0137   004   i544   0   0   18   4 -1   0003   0    0   0a
i534    1   0137   004   i534   0   0   19   4 -1   0003   0    0   0a
*
* These are Intel devices that use an interrupt vector but do not
* have any "bdevsw" or "cdevsw" entry.
*
debug   1     0     0    dbg    0   0    0   1 -1   0001   0    0   0a
slave7  1     0     0    sl     0   0    0   1 -1   0007   0    0   0a
*
* The following devices must not be specified in the system description
* file (xenixconf). These are pseudo drivers and the clock driver.
*
memory  0    06   0324   mm     0   -1   1   1 0    0      0    0   0a
tty     0   027   0324   sy     0   -1   12  1 0    0      0    0   0a
clock   1   000   0321          0   -1   -1  1 0    0      0    0   0a
$$$
```

Figure 6-1. Device Table from **sys/conf/master** (Continued)

Field 1, **name,** is the name of the device. The name begins in column 1 and is from 1 to 8 characters long. This name must be the same as the name used to identify the device in the **xenixconf** file. Intel devices are customarily identified as **ixxx**, e.g., **i534** for the iSBC 534 board. If limited to 4 characters, Field 1 can be identical to Field 5, **hndler.**

Field 2, **vsiz,** is the number of interrupt levels used by the device driver. Typically, each board uses a separate interrupt level. This number should be less than or equal to the number of interrupt levels specified in Fields 11 to 14 (4 levels maximum). Some drivers, such as that for the **i544,** use only a single interrupt level even if multiple boards are present; such a driver must poll the boards to determine the source of each interrupt. If a device does not use interrupts, then Field 2 is 0. A "virtual" device such as a RAM disk is an example of a device that does not use interrupts.

Field 3, **msk,** is an octal bit mask indicating which standard driver routines are present:

0100	**init** routine present
0020	**open** routine present (else replace with **nulldev**)
0010	**close** routine present (else replace with **nulldev**)
0004	**read** routine present (else replace with **nodev**)
0002	**write** routine present (else replace with **nodev**)
0001	**ioctl** routine present (else replace with **nodev**)

Note that neither the **intr** routine nor the **strategy** routine of block drivers is listed. The **intr** routine must be provided for all drivers that use interrupts. The **strategy** routine is mandatory for all block drivers. You can form the bit mask for your device by taking the mask values for all the routines present in your driver and ORing them. For example, for a line printer driver that provided all routines except **read** and **ioctl**, the mask value would be 0132. The kernel routine **nodev** replaces missing **read**, **write**, or **ioctl** routines. **nodev** indicates an error if it is called. The kernel routine **nulldev** replaces missing **open** or **close** routines in the **cdevsw** or **bdevsw** tables. **nulldev** does nothing when called, simply returning to its caller.

Field 4, **typ**, is an octal bit mask indicating device type and some miscellaneous information:

0200 Only one specification of the device is allowed; i.e., only one line in **master**'s device table can refer to the device.

0040 The device does *not* use interrupts.

0020 The device is *required* in the configuration. A required device is always included by the **config** program and must *not* be specified in the **xenixconf** file.

0010 The device provides a block interface.

0004 The device provides a character interface.

You can form the bit mask to specify for your device by taking all the mask values that apply to your driver and ORing them. Terminals and simple character devices have type 004. Disks, which normally support a "raw" character interface as well as a block interface, have type 014. A RAM disk might not need a character interface and could have type 010.

Field 5, **hndlr**, is the value that is prefixed to the standard routine names to produce the routine names used in your driver. For example, if **lp** is the value of the **hndlr** field for your device, your routine names *must* be **lpinit**, **lpopen**, etc. The prefix can be from 1 to 4 characters in length. The prefix must begin with a letter, and the characters in the prefix must be limited to those allowed in C identifiers. The prefix is used to generate the routine names in the switch tables **dinitsw**, **cdevsw**, **bdevsw**, and **vecintsw**. You can reduce confusion if your prefix is the same as the device name in Field 1.

Field 6, **na**, is not used and should be **0**.

Field 7, **bmaj**, is the major number used for the device's block interface. If the device does not have a block interface, the field is unused and typically **0** or **-1**. A major number of zero is allowed.

Field 8, **cmaj**, is the major number used for the device's character interface. If the device does not have a character interface, the field is unused and typically **0** or **-1**. A major number of zero is allowed.

For devices with both block and character interfaces, the same major number is typically used for both interfaces. While the block and character major numbers for a device can be different, it is recommended that they be the same.

Field 9, **#**, is the maximum number of boards supported by the device driver that may be present in the system.

Field 10, **na**, is not used and should be **-1**.

Fields 11, 12, 13, and 14 together contain up to four octal interrupt levels used by the driver. Unused interrupt levels should be zero; zero is not allowed as a valid interrupt level. Levels should be in the range 1-0377 octal (1-255 decimal). The levels specified must not conflict with those used by other devices and *must* be the same as those actually used by the hardware. If only one level is specified, use Field 11; if two levels are specified, use Fields 11 and 12, etc.

The letter **a** immediately follows field 14 because the **config** program does not allow a newline to immediately follow the field list. Any other character can be used, but 'a' is traditional.

The device table is organized as follows:

1. All block devices

2. All character devices

3. Special devices (e.g., **debug**) that use only an interrupt level and have no **bdevsw** or **cdevsw** entry

4. Pseudo devices that must not be specified in the **xenixconf** file (e.g., **tty**)

The line containing **$$$** terminates the device table. Subsequent sections of **master** give the line discipline table (for terminals), the alias table, and the tunable parameters table. All these are included in the **master** listing in Appendix G.

There is a tradeoff in choosing a major number for your device. If you choose a small major number, close to the numbers now being used by Intel, you may conflict with future Intel usage as Intel supports more devices. Of course, such a conflict is very easy to resolve, since your driver code should not depend on the major number in any way and only your **master** entries would need to be revised. If you choose a large major number, you expand the size of the switch table(s), wasting memory with null entries in those tables. Major numbers 7, 8, 9, 10, and 11 are available (in Release 3 of XENIX 286) for customer block devices or future Intel devices. Major numbers 20 and above are available (in Release 3 of XENIX 286) for any customer device or future Intel devices.

The name, prefix, major numbers, and interrupt levels for your device should be distinct from those used by other devices. The only exception to this rule is that a conflict between two devices is allowed if only one of them is ever included in the configuration via the **xenixconf** file. For example, the **i534** driver and the **i544** driver use the same interrupt level, thus only one of them may be included in your XENIX configuration (unless you modify the **master** file to place one of these devices at a different interrupt level and modify the hardware jumpers accordingly).

Editing xenixconf

You must add one line about your driver to the file **sys/conf/xenixconf.** This is a text file and can be edited with any XENIX editor. **xenixconf** specifies exactly which of the devices described in **master** are to be included in the new kernel that you are building. **master** must be changed only when adding a new driver or changing an interrupt level. **xenixconf** is changed when you add or remove devices. **xenixconf** begins with a block of comments, which have asterisks (*) in column 1. The device list follows the comment block, e.g.:

```
*           Devices
*
i215        1
i534        0
i544        0
i188        1
i74         1
lp          1
ramd        0
debug       0
root        i215  1
pipe        i215  1
swap        i215  2  1  1188
```

Each device entry consists of a device name and an *include flag.* The flag is 1 if the driver should be included in the system (i.e., there is hardware for it to support). Otherwise, the flag is 0. If the flag is 0, then the named driver is *not* included in the configuration, regardless of any entry in **master.** For example, in the configuration specified above, the **i534, i544, ramd,** and **debug** devices are not included. Note that if a device defined in **master** is not listed at all in **xenixconf,** that is equivalent to listing the device with a flag value of 0.

Entries in the **xenixconf** device list are in no particular order; you can insert your device at any point.

The lines with the names **root, pipe,** and **swap** are not new devices. These lines appear at the end of the device list and name devices used for system purposes. In the configuration specified above, the root file system uses the **i215** device, which is also used for swap space and pipe space.

A sample **xenixconf** listing is in Appendix H of this manual.

Editing the makefiles

The next editing task in adding a driver to the configuration is the revision of two makefiles: **sys/cfg/makefile** and **sys/io/makefile.** Both are text files and can be changed with any of the editors. Both are less than a page long and must be changed in only one place. In both files, the changes are made in the line that begins **OBJS= ;** you need to delete old object file names and add the name of an object file to each list.

In **sys/cfg/makefile,** delete the names of any object modules for which there is not a corresponding source file in **sys/cfg;** these object modules have already been generated and added to **lib_ioc.** Then add the name of the object file for your driver's configuration data structures: **cxxx.o.**

In **sys/io/makefile,** delete the names of any object modules for which there is not a corresponding source file in **sys/io;** these object modules have already been generated and added to **lib_io.** Then add the name of the object file for your main driver code: **ixxx.o.**

Note that simple file names, not path names, are used. Names in the lists are separated by spaces. If the list extends across more than one line, all lines but the last must end with a backslash (\) immediately followed by the newline character. The object files named in the list may not exist yet; the purpose of makefiles is to automate the process of producing such derived files as needed. Makefiles and the **make** program are described in the chapter "**make:** Program Maintainer" in the *XENIX 286 Programmer's Guide*.

Making a New Kernel

To make a new XENIX kernel, follow these steps:

1. Change your directory to **sys/cfg** and execute the **make** command (with no arguments). This compiles new or changed driver configuration files and adds them to the library of driver configuration structures, **sys/cfg/lib_ioc.**

2. Change your directory to **sys/io** and execute the **make** command (with no arguments). This compiles new or changed driver code and adds the resulting object files to the library of driver object modules, **sys/io/lib_io.**

3. Change your directory to **sys/conf** and execute the command

   ```
   make xenix
   ```

   This **make** command calls the **config** program that constructs the C program file **c.c** using the information in **master** and **xenixconf. c.c** is compiled and linked with the other files and libraries in the kernel. The new bootable kernel is placed in the file **sys/conf/xenix. c.c** contains the device switch tables, **dinitsw, cdevsw, bdevsw,** and **vecintsw.** A sample **c.c** listing is in Appendix I of this manual.

The new kernel must be placed in the root directory, **/**, before it can be used. You should *not* overwrite your present kernel, **/xenix**, because you do not know whether the new kernel will work or not. You should use a new file name, such as **/xenix.test**, when you move your new kernel to the root directory.

Making the Device Special File

For programs or commands to access your device, one or more special files must exist for it in the **/dev** directory. A special file can be created with the **mknod** command, with the form

```
/etc/mknod name c major minor
```

or

```
/etc/mknod name b major minor
```

where *name* is the new file name. **c** is specified for character interfaces; **b** is specified for block interfaces. A single special file can support either a character or block interface, but not both. For devices with both interfaces, two special files are needed to access the different interfaces. Finally, the major number of the device and the minor number of the new special file are specified. Note that a separate special file must be created for *each* minor number. The major and minor numbers may be specified in either decimal or octal. For example, to create special files for two line printers:

```
/etc/mknod /dev/lp0 c 7 0
/etc/mknod /dev/lp1 c 7 1
```

While there are no formal naming conventions for device special files, you may want to consult the existing nodes in **/dev** and name your files in the same way. For example, nodes for raw (character) interfaces to block devices begin with 'r'; nodes for partitions on block devices have names such as 'w0' (entire disk), 'w0a', 'w0b', and 'w0c' (partitions for parts of the disk).

You should edit the file **/dev/makefile** and add the **mknod** commands for your device. Then make **/dev** your directory and run **make**. This will execute any needed **mknod** commands to create device special files. Only the super-user can create device special files.

Adding Terminal Information

If you are adding a terminal device driver to your system, you may have to add information to two system files: **/etc/ttytype** and **/etc/ttys**. Refer to the section "Adding a Terminal" in the chapter "Tailoring the Environment" in the *XENIX 286 System Administrator's Guide* for more information on this subject. If a new type of terminal is being connected to your driver, then you may have to add information to the file **/etc/termcap**, described in "Files" in the *XENIX 286 Reference Manual*.

Executing the New Kernel

Before booting the new kernel, be certain that any hardware required by your driver is properly installed. You can then boot your new kernel, as described in the *XENIX 286 Installation and Configuration Guide*.

Deleting a Device Driver

To delete a driver from your system, follow these steps:

1. Change the include flag for the driver in **xenixconf** to **0**.

2. Make a new kernel, as described in this chapter.

3. Delete any device special files that refer to the driver (using the **rm** command). (This step is not mandatory; you can leave the special files in **/dev** if they may be used again in the future.)

4. Reboot your system with the new kernel.

The driver will not be present in the new kernel. However, the driver source and object code will still be present and maintained by the makefiles. If you delete the three driver files, you should also remove references to the driver in **sys/cfg/makefile** and **sys/io/makefile**. If you delete the device special files for the driver, then you should remove the corresponding **mknod** commands in **/dev/makefile**.

intel®

APPENDIX A
MEMORY-MAPPED I/O FOR DRIVERS

This appendix describes how device drivers can read and write device registers that are mapped into the memory address space outside of the kernel data segment. This is an alternative to mapping device registers into the I/O port address space, which supports 2^{16} port addresses. The memory address space can be used to allow more efficient control of device registers with a greater variety of iAPX 286 instructions. For example, instructions to increment a memory location or to move a block of bytes or words can be applied directly to device registers mapped into memory. Memory-mapped I/O is used by some Intel-supplied drivers, including the iSBC 544 and iSBC 188/48 drivers.

Small Model Kernel

The XENIX 286 kernel (which includes all device driver code) is implemented as a small model program with a single data segment. The single kernel data segment increases speed and reduces storage requirements in the kernel, because data pointers in the kernel are simply 16-bit offsets.

There is not enough space in the kernel data segment for memory-mapped I/O devices, so separate segments must be created to frame the device registers and to be used when accessing the device. After such a segment is created, the peek and poke routines described in this appendix can be used to access the device registers.

Creating the Segment Descriptor

An available descriptor in the iAPX 286 Global Descriptor Table (GDT) is allocated by calling **dscralloc,** which returns the selector for the descriptor. The descriptor must then be initialized by calling **mmudescr** with parameters that specify the selector, the segment's physical base address, the segment's length, and access rights for the segment.

```
unsigned dscralloc();
/*
Allocates an available descriptor in the Global Descriptor Table for
the use of the caller. Returns the selector for the allocated
descriptor. Does NOT initialize the descriptor.
*/
```

If no more descriptors are available, a major kernel error results. The total number of descriptors available in the GDT is a configuration option.

```
mmudescr(selector, paddr, ceil, access)
unsigned    selector;    /* selector for descriptor being overwritten    */
long        paddr;       /* contains 24-bit physical base address        */
unsigned    ceil;        /* segment length minus one                     */
int         access;      /* low byte of access is used as access
                            rights byte of the descriptor.               */
```

In calling **mmudescr,** use an **access** value of 0x92. Bit 7 is set to indicate that the segment is present. Bits 6 and 5 are clear to indicate maximum privilege level (0), appropriate to a segment being accessed from kernel code. Bit 4 is set to indicate a segment descriptor. Bit 3 is clear to indicate a data descriptor (versus a code descriptor). Bit 2 is clear to indicate that any expansion of the segment will be up, not down (not applicable in this case). Bit 1 is set to indicate that the segment is writable. Bit 0 is clear indicating that the segment has not yet been accessed.

The following code illustrates the creation of a segment to frame a device that maps 16K bytes of memory starting at physical address 0xfe0000:

```
unsigned  selector = dscralloc();
mmudescr(selector, 0xfe0000, 0x3fff, 0x92);
```

The Peek Routines

The following peek routines are part of the XENIX 286 kernel:

```
peek(offset, selector, count, addr)
unsigned    offset;      /* offset into source data segment              */
unsigned    selector;    /* segment selector for source data segment     */
unsigned    count;       /* number of bytes to transfer (0-65535)        */
char        *addr;       /* destination address in kernel data segment
                            (short pointer, offset only)                 */
/*
Copies count bytes beginning at offset in the data segment specified
by selector, to an area of the same size beginning at addr in the
kernel data segment.  count can be 0, in which case no bytes are
copied.
*/


int peekb(offset, selector)
unsigned    offset;      /* offset into source data segment              */
unsigned    selector;    /* segment selector for source data segment     */
/*
Reads one byte at the specified offset in the specified data segment.
The value read is returned as an int value in the range 0-255.
*/
```

```
int peekw(offset, selector)
unsigned    offset;     /* offset into source data segment          */
unsigned    selector;   /* segment selector for source data segment */
/*
Reads one word (two bytes) at the specified offset in the specified
data segment.  The value read is returned as an int.
*/
```

For single bytes or words, the **peekb** or **peekw** routines should be used instead of **peek.**

The Poke Routines

The following poke routines are part of the XENIX 286 kernel:

```
poke(offset, selector, count, addr)
unsigned    offset;     /* offset into destination data segment     */
unsigned    selector;   /* segment selector for destination segment */
unsigned    count;      /* number of bytes to transfer (0-65535)    */
char        *addr;      /* source address in kernel data segment
                           (short pointer, offset only)             */
/*
Copies count bytes beginning at addr in the kernel data segment, to
an area beginning at offset in the data segment specified by selector.
count can be 0, in which case no bytes are copied.
*/


pokeb(offset, selector, value)
unsigned    offset;     /* offset into destination data segment     */
unsigned    selector;   /* segment selector for destination segment */
int         value;      /* int with low byte to be poked            */
/*
Writes the low byte of value (value & 0xff) at the specified offset
in the specified data segment.
*/


pokew(offset, selector, value)
unsigned    offset;     /* offset into destination data segment     */
unsigned    selector;   /* segment selector for destination segment */
int         value;      /* int (word) value to be poked             */
/*
Writes the word value at the specified offset in the specified
data segment.
*/
```

For single bytes or words, the **pokeb** or **pokew** routines should be used instead of **poke.**

intel®

APPENDIX B
CONVERTING DRIVERS FROM RELEASE 1 TO RELEASE 3 OF XENIX 286

This appendix describes how changes in the device driver interface affect the conversion of drivers. The changes made to the device driver interface in XENIX Release 3 represent improvements over the Release 1 version. Relatively few changes are required to convert a block driver from Release 1 to Release 3. While significant changes have been made to character devices (particularly terminal drivers), conversion should not require a major rewrite of the code.

The changes in character device drivers are primarily the result of the more sophisticated controllers available on the market. New hardware and better firmware have reduced the workload of the machine-dependent line discipline routines. They are, therefore, simplified under Release 3. In general, as a device increases in functionality, the driver-device interface becomes more complex: the device requires more information, and the driver must provide it. Consequently, the driver routines in Release 3 are expanded, and the **tty** structure has been altered to hold more information.

Block device driver changes are fairly minor and result from the fact that XENIX Release 3 supports large model programs. The changes that are included affect the static buffer header associated with each device (a cosmetic upgrade) and the way in which the driver addresses memory.

Terminal Drivers

tty Structure

Significant changes have been made in this area. Many fields previously present in the product have been eliminated or replaced; new fields have been added.

Table B-1 shows fields that have changed in the **tty** structure. Note that some have been replaced by new fields in the Release 3 version, while others have been eliminated.

Table B-2 shows the Release 3 fields that did not exist in the Release 1 **tty** structure. Some of these new fields contain information that was previously contained in a different Release 1 field. Others are entirely new; they have no Release 1 counterpart and contain no Release 1 information. The **tty** structure is defined in the include file **tty.h**, which is listed in Appendix C of this manual.

An example of an actual Release 3 terminal driver appears in Chapter 4, "Terminal Drivers." Studying the example code in Chapter 4 may help you to understand the points made about converting terminal drivers in this appendix.

Table B-1. Changed **tty** Fields

Release 1 Field	Comment	Release 3 Field	Comment
int (*t_oproc)	Pointer routine to start output	**int (*t_proc)()**	Pointer to new routine that starts input and output, and changes the **tty** structure if necessary. The driver writer must write this new routine.
()int (*t_iproc)()	Pointer routine to start input		
struct chan *t_chan	Destination channel for multiplexed files	(eliminated)	Multiplexed files not supported.
caddr_t t_linep	Auxiliary line discipline pointer	(eliminated)	No longer needed.
caddr_t t_addr	Device address	**short t_addr**	Device number.
dev_t t_dev	Device number	(eliminated)	**t_addr** field used.
short t_flags	**octl** models	(eliminated)	Replaced by several fields.
short t_2state	Driver-specific state	(eliminated)	
char t_erase	Erase character	(eliminated)	Now bit-mapped in **char** control array field.
char t_kill	Kill character	(eliminated)	Now bit-mapped in **char** control array field.
char t_ispeed	Input speed	(eliminated)	Both speed fields are replaced by a single bit-mapped speed in the new control mode field.
char t_ospeed	Output speed	(eliminated)	
union t_un	Extended **ioctl** control structures	(eliminated)	Information encoded in new mode fields.

Table B-2. New **tty** Fields

New Field	Comment
ushort t_iflag	Input modes; values for this field are located in the new file **termio.h.** Release 1 drivers were concerned with very few input modes; most were not an option. This is an expanded capability of Release 3.
ushort t_oflag	Output modes; values for this field are also located in the new file **termio.h.** In Release 1, both input and output mode values were default (assumed) or ignored in the line discipline routines. However, some controllers need this information, and these fields allow a user to set the modes.
ushort t_cflag	Control modes; values for this field are also located in the new file **termio.h.** This field serves the same purpose that the **tc** structure together with the **ttiocb** structure served in Release 1: it changes the **tty** characteristics (e.g., baud rate).
ushort t_lflag	Line discipline modes; new to Release 3. These modes are used by the line discipline routines. Bits for this field are defined in **termio.h.**
ushort t_xflag	External protocol modes; new to Release 3. These allow different protocols. Bits for this field are defined in **termio.h.**
chart t_row	Current row; it may be useful to some drivers to know which line a user is on if the last line has been reached.
struct tty*t_chan	Pointer to multidrop channels.

Changes to Routines

Under Release 1, the routines required to interface with the XENIX kernel included

ixxxinit
ixxxopen
ixxxclose
ixxxstart
ixxxread
ixxxwrite
ixxxintr
ixxxioctl

The **ixxxstart** routine is no longer a required interface routine; it is now an optional internal routine. Replacing and expanding considerably on the function of **ixxxstart** is **ixxxproc.** It is a required routine, and a field in the **tty** structure **(tp->t_proc)** holds a pointer to it. Several line discipline routines including **ttyclose, ttyflush, canon, ttrstrt, ttyread,** and **ttywrite** call **ixxxproc** to effect some change on the output.

The parameters to **ixxxproc** include **tp** and **cmd. tp** is a pointer to the **tty** structure; **ixxxstart** takes **tp** as its only parameter. Thus, with this parameter alone, **ixxxproc** could accomplish what **ixxxstart** does. However, the expanded capability of **ixxxproc** is reflected in its second argument, **cmd,** which dictates what action--if any--**ixxxproc** should take. The commands that must be handled and their meanings are listed in Table B-3.

Table B-3. **ixxxproc** Commands

Command	Function
T_TIME	Time delay for outputting a break has finished.
T_WFLUSH	Flush output queue.
T_RESUME	Output was stopped or someone is waiting for the output queue to drain.
T_OUTPUT	Start output.
T_SUSPEND	Stop output on this line.
T_BLOCK	Block input.
T_UNBLOCK	Start input.
T_RFLUSH	Someone is waiting to flush the input queue.
T_BREAK	Send a break.

In Release 1, these functions were handled as machine-independent features. Since they are truly machine-dependent, they are now included as a user-written routine. In converting a Release 1 driver to Release 3, **ixxxstart** may be made a routine internal to **ixxxproc**, and code to handle the other commands listed in Table B-3 would have to be written.

The new **ixxxproc** procedure is now called with two arguments:

```
tp   /* pointer to tty structure */
cmd /* user command to change output */
```

An example of code for an **ixxxproc** procedure is contained in Chapter 4, "Terminal Drivers."

In Release 3, **ixxxioctl** is called by the kernel with the first two arguments swapped:

```
ixxxioctl(cmdarg, dev, addr, flag)
```

That is, in Release 1, **ixxxioctl** was called in this order:

```
ixxxioctl(dev, cmdarg, addr, flag)
```

Note that the **cmdarg** argument is a **long** type under Release 3. Formerly, it was an **int**.

In addition to the file **tty.c**, which under Release 1 contained all the line discipline routines, there now exists a file **tt0.c**. Some of the line discipline routines are located in **tty.c**, and the others are located in **tt0.c**. (This information is useful only to source code customers.)

Another new Release 3 file is **ttold.h**. It contains all the Release 1 structure definitions that Release 3 requires in order to maintain UNIX Version 7 compatibility. These definitions include the **ioctl** user structures, so that user programs written under Release 1 will be source-compatible under Release 3. If the **ttold.h** file is included in a Release 3 driver and all other Release 3 changes have been made, then user programs should be compatible.

Line Discipline Routines

The line discipline routines (those accessible by the driver) in Release 1 have been replaced by a new set of routines in Release 3, as Table B-4 indicates. Some of the names have remained the same, and the functionality has changed only in that the new driver routine **ixxxproc** does most of the work these routines did in Release 1. These Release 3 routines are listed in the **linesw** table. (Note that the arrangement of routines in Table B-4 does not indicate a correspondence between all pairs of routines. E.g., **l_rend** does not correspond to **l_input.**)

Table B-4. Line Discipline Routines

Release 1	Release 3
l_open	l_open
l_close	l_close
l_read	l_read
l_write	l_write
l_ioctl	l_ioctl
l_rint	l_output
l_rend	l_input
l_meta	l_mdmint
l_modem	

The tty.h File

The most obvious change to **tty.h** is that much of its information has been expanded and moved into two new files, **ttold.h** and **termio.h.** The **ttold.h** file contains structures as defined under XENIX 286 Release 1; it allows compatibility with UNIX Version 7. The **termio.h** file contains the bit values defined for the four new mode fields located in the **tty** structure:

input modes	**(tp->t_iflag)**
output modes	**(tp->t_oflag)**
control modes	**(tp->t_cflag)**
line disciplines	**(tp->t_lflag)**

Several fields contained in the **tty** structure under Release 1 have been eliminated and replaced by these four mode fields. The values for the new mode fields encode much more information than the Release 1 fields, reflecting the fact that the driver is handling more than it did in the previous release.

The Release 1 fields that have been replaced include **t_flags, t_state, t_2state, t_erase, t_kill, t_char, t_ispeed,** and **t_ospeed.**

A comparison between the Release 1 structure fields and the bit values for the Release 3 **tty** structure fields reveals that all Release 1 information is still present under Release 3. The form has simply changed as a result of the need to keep track of more information. The values for the new Release 3 mode fields are displayed in Tables B-5, B-6, B-7, and B-8. These values are defined in **termio.h**, which is listed in Appendix D of this manual.

Table B-5. Input Modes Describing Basic Terminal Input Control

Input Modes	Octal Values	Comments
IGNBRK	0000001	Ignores break condition.
BRKINT	0000002	Signals interrupt on break.
IGNPAR	0000004	Ignores characters with parity errors.
PARMRK	0000010	Marks parity errors.
INPCK	0000020	Enables input parity check.
ISTRIP	0000040	Strips characters.
INLCR	0000100	Maps newline to carriage return on input.
IGNCR	0000200	Ignores carriage return.
ICRNL	0000400	Maps carriage return to newline on input.
IUCLC	0001000	Maps uppercase to lowercase on input.
IXON	0002000	Enables start/stop output control.
IXANY	0004000	Enables any character to restart output.
IXOFF	0010000	Enables start/stop input control.

Table B-6. Output Modes Specifying System Treatment of Output

Output Modes	Octal Values	Comments
OPOST	0000001	Postprocesses output.
OLCUC	0000002	Maps lowercase to uppercase on output.
ONLCR	0000004	Maps newline to carriage return-newline on output.
OCRNL	0000010	Maps carriage return to newline on output.
ONOCR	0000020	No carriage return output at column 0.
ONLRET	0000040	Newline performs carriage return function.
OFILL	0000100	Uses fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Selects newline delays.
CRDLY	0003000	Selects carriage return delays.
TABDLY	0014000	Selects horizontal tab delays.
BSDLY	0020000	Selects backspace delays.
VTDLY	0040000	Selects vertical tab delays.
FFDLY	0100000	Selects formfeed delays.

Table B-7. Control Modes Describing Hardware Control of the Terminal

Control Modes	Octal Values	Comments
CBAUD	0000017	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud
B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
EXTA	0000016	External A
EXTB	0000017	External B
CSIZE	0000060	Character size:
CS5	0	5 bits
CS6	0000020	6 bits
CS7	0000040	7 bits
CS8	0000060	8 bits
CSTOPB	0000100	Sends 2 stop bits, else 1
CREAD	0000200	Enables receiver
PARENB	0000400	Parity enable
PARODD	0001000	Odd parity, else even
HUPCL	0002000	Hangs up on last close
CLOCAL	0004000	Local line, else dial-up

Table B-8. Line Discipline Modes Used to Control Terminal Function

Line Discipline Modes	Octal Values	Comments
ISIG	0000001	Enables signals
ICANON	0000002	Canonical input (erase and kill processing)
XCASE	0000004	Canonical upper/lower presentation
ECHO	0000010	Enables echo
ECHOE	0000020	Echoes erase character as backspace-space-backspace
ECHOK	0000040	Echoes newline after kill character
ECHONL	0000100	Echoes newline
NOFLSH	0000200	Disables flush after interrupt or quit

Within **termio.h** is one defined structure called **termio.** It is the **ioctl** control packet; that is, it contains all information needed by the **ioctl** routine. That information includes all the mode information listed in the four preceding tables plus information on which set of line routines to use, the external protocol modes, and settings for control characters that were located in the Release 1 **tty** structure (erase, kill, etc.).

Block Device Drivers

Buffer Changes

In Release 1, the buffer header was defined in the file **buf.h,** and all buffer headers, including the static buffer header for each device, were of this format. The static buffer header did not use most of the fields as defined in **buf.h** because most of them dealt with I/O request information. (The static buffer header merely acts as a queue header.)

In Release 3, defining a new format for the static buffer header distinguishes the static buffer header from a regular buffer header used to make I/O requests. The new format is defined in **iobuf.h** and has fields appropriate to a queue header. In block drivers, then, the static buffer header is declared as **iobuf** rather than **buf.** (In Release 1, the static buffer header was usually declared by the name **bufh.**) **iobuf.h** is listed in Appendix F of this manual and is also described in Chapter 5, "Block Drivers."

Addressing

In Release 1, the buffer header as defined in **buf.h** contained several fields used to address the device. These fields included a union of **caddr_t b_addr** and **char b_xmem.** In Release 3, these fields have been replaced with **p_addr,** a single field representing a 24-bit physical address. Addressing is now much simpler. Wherever the routine **physaddr** was used in Release 1 to put together a physical address, **bp->p_addr** can be used directly.

The **cmdarg** argument in the **ioctl** routine was a short pointer in Release 1 (where short means 16-bit offset only). Because Release 3 is large-model (and has many data segments), this argument is now a **long** pointer. Recall that **cmdarg** is a pointer to a structure in space. Under Release 1, the system routines **fuword** and **fubyte** were used to access the fields in the structure. With Release 3, the system routine **copyin** can be used to make a local copy of the structure, which is more efficient for accessing fields. **copyin** is described in Chapter 2, "Driver Fundamentals."

intel®

APPENDIX C
tty.h INCLUDE FILE

This appendix lists the **tty.h** include file used by character drivers, including terminal drivers. Note that **tty.h** includes the include file **termio.h,** which is Appendix D of this manual.

```
/*
 * THIS FILE CONTAINS CODE WHICH IS DESIGNED TO BE
 * PORTABLE BETWEEN DIFFERENT MACHINE ARCHITECTURES
 * AND CONFIGURATIONS. IT SHOULD NOT REQUIRE ANY
 * MODIFICATIONS WHEN ADAPTING XENIX TO NEW HARDWARE.
 */


#include        "termio.h"

/*
 * A clist structure is the head of a linked list queue of characters.
 * The routines getc* and putc* manipulate these structures.
 */
struct clist {
        int         c_cc;               /* character count */
        struct      cblock      *c_cf;  /* pointer to first */
        struct      cblock      *c_cl;  /* pointer to last */
};

/*
 * A tty structure is needed for each UNIX character device that
 * is used for normal terminal IO.
 */
struct tty {
        struct      clist       t_rawq;     /* raw input queue */
        struct      clist       t_canq;     /* canonical queue */
        struct      clist       t_outq;     /* output queue */
        struct      cblock      *t_buf;     /* buffer pointer */
        int         (*t_proc)();            /* routine for device functions */
        ushort      t_iflag;                /* input modes */
        ushort      t_oflag;                /* output modes */
        ushort      t_cflag;                /* control modes */
        ushort      t_lflag;                /* line discipline modes */
        ushort      t_xflag;                /* external protocol modes */
        short       t_state;                /* internal state */
        short       t_pgrp;                 /* process group name */
        char        t_line;                 /* line discipline */
        char        t_delct;                /* delimiter count */
        char        t_col;                  /* current column */
        char        t_row;                  /* current row */
```

```
        uchar_t     t_cc[NCC + 2];          /* settable control chars */
        short       t_addr;                 /* v7 compatibility */
        struct      tty     *t_chan;        /* multi-drop channels, pointer to */
};
/*
 * The structure of a clist block
 */
#define CLSIZE  24
struct      cblock {
        struct      cblock      *c_next;
        char        c_first;
        char        c_last;
        char        c_data[CLSIZE];
};

extern      struct      cblock      cfree[];
extern      struct      cblock      *getcb();
extern      struct      cblock      *getcf();
extern      struct      clist       ttnulq;

struct      chead   {
        struct      cblock      *c_next;
        int         c_size;
};
extern      struct      chead       cfreelist;

struct      inter   {
        int         cnt;
};

/* control characters */                    /* pick up from termio.h */

/* default control chars */                 /* pick up from termio.h */

#define TTIPRI          28
#define TTOPRI          29

/* limits */
extern      int         ttlowat[],  tthiwat[];
#define     TTYHOG      256
#define     TTXOLO      60
#define     TTXOHI      180

/* input modes */                           /* pick up from termio.h */

/* output modes */                          /* pick up from termio.h */

/* control modes */                         /* pick up from termio.h */

/* line discipline 0 modes */               /* pick up from termio.h */

/* default speed */                         /* pick up from termio.h */
```

```
/* Hardware bits */
#define     DONE        0200
#define     IENABLE     0100
#define     OVERRUN     040000
#define     FRERRO      020000
#define     PERROR      010000

/* Internal state */
#define     TIMEOUT     01          /* Delay timeout in progress */
#define     WOPEN       02          /* Waiting for open to complete */
#define     ISOPEN      04          /* Device is open */
#define     TBLOCK      010
#define     CARR_ON     020         /* Software copy of carrier-present */
#define     BUSY        040         /* Output in progress */
#define     OASLP       0100        /* Wakeup when output done */
#define     IASLP       0200        /* Wakeup when input done */
#define     TTSTOP      0400        /* Output stopped by ctl-s */
#define     EXTPROC     01000       /* External processing */
#define     TACT        02000
#define     ESC         04000       /* Last char escape */
#define     RTO         010000
#define     TTIOW       020000
#define     TTXON       040000
#define     TTXOFF      0100000

/* l_output status      */
#define     CPRES       1

/* device commands      */
#define     T_OUTPUT    0
#define     T_TIME      1
#define     T_SUSPEND   2
#define     T_RESUME    3
#define     T_BLOCK     4
#define     T_UNBLOCK   5
#define     T_RFLUSH    6
#define     T_WFLUSH    7
#define     T_BREAK     8
```

intel®

APPENDIX D
termio.h INCLUDE FILE

This appendix lists the **termio.h** include file used by terminal drivers. **termio.h** is included by the include file **tty.h,** and terminal drivers can just include **tty.h** and will still include all the definitions in **termio.h. tty.h** is listed in Appendix C of this manual.

```
/*
 * THIS FILE CONTAINS CODE WHICH IS DESIGNED TO BE
 * PORTABLE BETWEEN DIFFERENT MACHINE ARCHITECTURES
 * AND CONFIGURATIONS. IT SHOULD NOT REQUIRE ANY
 * MODIFICATIONS WHEN ADAPTING XENIX TO NEW HARDWARE.
 */
/*
 * Modification history
 * l001          4/30/84          comment
 *                                Added definitions for baud rates higher than
 *                                9600 baud. Also added definitions for extra flag
 *                                field.
  */


#define     NCC         8

/* control characters */
#define     VINTR       0
#define     VQUIT       1
#define     VERASE      2
#define     VKILL       3
#define     VEOF        4
#define     VEOL        5
#define     VMIN        4
#define     VTIME       5
#define     VCEOF       NCC         /* RESERVED true EOF char (V7 compatability) */
#define     VCEOL       (NCC + 1)   /* RESERVED true EOL char */

#define     CNUL        0
#define     CDEL        0377
/* default control chars */
#define     CESC        '\\'
#define     CINTR       0177        /* DEL */
#define     CQUIT       034         /* FS, cntl | */
#define     CERASE      '\010'      /* backsp */
#define     CKILL       '\025'      /* cntl u */
#define     CEOF        04          /* cntl d */
#define     CSTART      021         /* cntl q */
#define     CSTOP       023         /* cntl s */
```

```
/* input modes */
#define     IGNBRK      0000001
#define     BRKINT      0000002
#define     IGNPAR      0000004
#define     PARMRK      0000010
#define     INPCK       0000020
#define     ISTRIP      0000040
#define     INLCR       0000100
#define     IGNCR       0000200
#define     ICRNL       0000400
#define     IUCLC       0001000
#define     IXON        0002000
#define     IXANY       0004000
#define     IXOFF       0010000

/* output modes */
#define     OPOST       0000001
#define     OLCUC       0000002
#define     ONLCR       0000004
#define     OCRNL       0000010
#define     ONOCR       0000020
#define     ONLRET      0000040
#define     OFILL       0000100
#define     OFDEL       0000200
#define     NLDLY       0000400
#define     NL0         0
#define     NL1         0000400
#define     CRDLY       0003000
#define     CR0         0
#define     CR1         0001000
#define     CR2         0002000
#define     CR3         0003000
#define     TABDLY      0014000
#define     TAB0        0
#define     TAB1        0004000
#define     TAB2        0010000
#define     TAB3        0014000
#define     BSDLY       0020000
#define     BS0         0
#define     BS1         0020000
#define     VTDLY       0040000
#define     VT0         0
#define     VT1         0040000
#define     FFDLY       0100000
#define     FF0         0
#define     FF1         0100000
```

```
/* control modes */
#define     CBAUD       0000017
#define     EXBAUD      0070000
#define     B0          0
#define     B50         0000001
#define     B75         0000002
#define     B110        0000003
#define     B134        0000004
#define     B150        0000005
#define     B200        0000006
#define     B300        0000007
#define     B600        0000010
#define     B1200       0000011
#define     B1800       0000012
#define     B2400       0000013
#define     B4800       0000014
#define     B9600       0000015
#define     B19200      0000016
#define     B38400      0000017
#define     B51800      0010017
#define     B76800      0020017
#define     EXTA        0000016
#define     EXTB        0000017
#define     CSIZE       0000060
#define     CS5         0
#define     CS6         0000020
#define     CS7         0000040
#define     CS8         0000060
#define     CSTOPB      0000100
#define     CREAD       0000200
#define     PARENB      0000400
#define     PARODD      0001000
#define     HUPCL       0002000
#define     CLOCAL      0004000

/* line discipline 0 modes */
#define     ISIG        0000001
#define     ICANON      0000002
#define     XCASE       0000004
#define     ECHO        0000010
#define     ECHOE       0000020
#define     ECHOK       0000040
#define     ECHONL      0000100
#define     NOFLSH      0000200
#define     XCLUDE      0100000     /* *V7* exclusive use */
```

```
/* external protocol modes */
#define     XLSIG       0000177     /* type of line signaling */
#define     RS232       0000000     /* RS 232 line */
#define     RS422       0000001     /* RS 422 line */
#define     RS485       0000002     /* RS 485 line */
#define     XHDLC       0001000     /* hdlc packet protocol */
#define     XSDLC       0002000     /* sdlc packet protocol */
#define     XBISC       0004000     /* bi-sync protocol */
#define     X25         0010000     /* CCITT x.25 packet protocol */
#define     XMTDP       0020000     /* multidrop device */

#define     SSPEED      13          /* default speed: 7 = 300, 13 = 9600 baud */

/*
 * Ioctl control packet
 */
struct termio {
            unsigned    short   c_iflag;        /* input modes */
            unsigned    short   c_oflag;        /* output modes */
            unsigned    short   c_cflag;        /* control modes */
            unsigned    short   c_lflag;        /* line discipline modes */
            char                c_line;         /* line discipline */
            uchar_t             c_cc[NCC];      /* control chars */
};
```

intel®

APPENDIX E
buf.h INCLUDE FILE

This appendix lists the **buf.h** include file, which is included by block drivers. Chapter 5, "Block Drivers", contains a more detailed description of the **buf** data structure.

```
/*
 * THIS FILE CONTAINS CODE WHICH IS DESIGNED TO BE
 * PORTABLE BETWEEN DIFFERENT MACHINE ARCHITECTURES
 * AND CONFIGURATIONS. IT SHOULD NOT REQUIRE ANY
 * MODIFICATIONS WHEN ADAPTING XENIX TO NEW HARDWARE.
 */



/*
 * Each buffer in the pool is usually doubly linked into 2 lists:
 * the device with which it is currently associated (always)
 * and also on a list of blocks available for allocation
 * for other use (usually).
 * The latter list is kept in last-used order, and the two
 * lists are doubly linked to make it easy to remove
 * a buffer from one list when it was found by
 * looking through the other.
 * A buffer is on the available list, and is liable
 * to be reassigned to another disk block, if and only
 * if it is not marked BUSY.  When a buffer is busy, the
 * available-list pointers can be used for other purposes.
 * Most drivers use the forward ptr as a link in their I/O active queue.
 * A buffer header contains all the information required to perform I/O.
 * Most of the routines which manipulate these things are in bio.c.
 */
struct buf
{
        int     b_flags;        /* see defines below */
        struct  buf *b_forw;    /* headed by d_tab of conf.c */
        struct  buf *b_back;    /*  "  */
        struct  buf *av_forw;   /* position on free list, */
        struct  buf *av_back;   /*    if not BUSY*/
        dev_t   b_dev;          /* major + minor device name */
        unsigned    b_bcount;   /* transfer count */
        paddr_t b_paddr;        /* physical address */
#define paddr(X)        X->b_paddr
        daddr_t b_blkno;        /* block # on device */
        char    b_error;        /* returned after I/O */
        unsigned    int b_resid;    /* words not transferred after error
                */
        ushort  b_cylin;        /* cylinder number for disk i/o
                queue */
};
```

```
extern      struct buf buf[];                   /* The buffer pool itself */
extern      struct buf bfreelist;               /* head of available list */
extern      struct buf *lp_p;                   /* Low priority pointer */
extern      int lp_count;                       /* Number of low priority buffers */
extern      int lp_wmark;                       /* Low priority water mark */
extern      char sabuf[][BSIZE];

#ifdef      BUFMAPOUT
long        bigetl();
#else
#define     bigetc(bp,cp)  (*(char *) (bp->b_paddr + cp))
#define     biget(bp,cp)  (*(int *) (bp->b_paddr + cp))
#define     bigetl(bp,cp)  (*(long *) (bp->b_paddr + cp))
#define     biputc(bp,cp,c)  (*(char *) (bp->b_paddr + cp) = c)
#define     biput(bp,cp,c)  (*(int *) (bp->b_paddr + cp) = c)
#define     biputl(bp,cp,c)  (*(long *) (bp->b_paddr + cp) = c)
#endif

paddr_t     bufbase;

/*
 * These flags are kept in b_flags.
 */
#define     B_WRITE   0         /* non-read pseudo-flag */
#define     B_READ    01        /* read when I/O occurs */
#define     B_DONE    02        /* transaction finished */
#define     B_ERROR   04        /* transaction aborted */
#define     B_BUSY    010       /* not on av_forw/back list */

#ifdef      DHISTO
/*
 * We are running out of bits in the buffer flags. There is only one
 * bit flag left which is 040000. Since B_MAP and B_PHYS are not used
 * I stold them for the DHISTO program. B_PHYS was set in mdep/physio
 * but never tested.
 */
#define     B_BMISS   020       /* Signifies a buffer miss, i.e went to disk */
#define     B_USERB   040       /* Signifies a user buffer */
#define     DH_MAX    8192      /* Maximum number of dhisto device data points */

#else

#define     B_PHYS    020       /* Physical IO potentially using UNIBUS map */
#define     B_MAP     040       /* This block has the UNIBUS map allocated */
#endif
```

```
#define     B_WANTED  0100      /* issue wakeup when BUSY goes off */
#define     B_AGE     0200      /* delayed write for correct aging */
#define     B_ASYNC   0400      /* don't wait for I/O completion */
#define     B_DELWRI  01000     /* don't write till block leaves available list */
#define     B_OPEN    02000     /* open routine called */
#define     B_STALE   04000
#define     B_CYLIN   010000    /* buffer contains a cyl grp header */
#define     B_LOWPRI  020000    /* Buffer contains low priority data */
#define     B_UAREA   0100000   /* add u-area to a swap operation */

/*
 * Fast access to buffers in cache by hashing.
 */

#define     bhash(d,b)          ((struct buf *)&hbuf[((int)d + (int)b)&v.v_hmask])

struct      hbuf
{
            int         b_flags;
            struct      buf *b_forw;
            struct      buf *b_back;
};

extern struct hbuf hbuf[];
```

intel®

APPENDIX F
iobuf.h INCLUDE FILE

This appendix lists the **iobuf.h** include file, which is included by block drivers. Chapter 5, "Block Drivers", describes the **iobuf** data structure in more detail.

```
/*
 * THIS FILE CONTAINS CODE WHICH IS DESIGNED TO BE
 * PORTABLE BETWEEN DIFFERENT MACHINE ARCHITECTURES
 * AND CONFIGURATIONS. IT SHOULD NOT REQUIRE ANY
 * MODIFICATIONS WHEN ADAPTING XENIX TO NEW HARDWARE.
 */



/*
 * Each block device has a iobuf, which contains private state stuff
 * and 2 list heads: the b_forw/b_back list, which is doubly linked
 * and has all the buffers currently associated with that major
 * device; and the d_actf/d_actl list, which is private to the
 * device but in fact is always used for the head and tail
 * of the I/O queue for the device.
 * Various routines in bio.c look at b_forw/b_back
 * (notice they are the same as in the buf structure)
 * but the rest is private to each device driver.
 */
struct iobuf
{
        int             b_flags;                /* see buf.h */
        struct  buf     *b_forw;                /* first buffer for this dev */
        struct  buf     *b_back;                /* last buffer for this dev */
        struct  buf     *b_actf;                /* head of I/O queue */
        struct  buf     *b_actl;                /* tail of I/O queue */
        dev_t           b_dev;                  /* major + minor device name */
        char            b_active;               /* busy flag */
        char            b_errcnt;               /* error count (for recovery) */
        physadr         io_addr;                /* csr address */
        int             io_s1;                  /* space for drivers to leave things
        */
        int             io_s2;                  /* space for drivers to leave things
                                                   */
};

#define     tabinit(dv,stat)                {0,0,0,0,0,makedev(dv,0),0,0,0,0,0,stat,0,0}
#define     NDEVREG     (sizeof(struct device)/sizeof(int))

#define     B_ONCE      01          /* flag for once only driver operations */
#define     B_TAPE      02          /* this is a magtape (no bdwrite) */
#define     B_TIME      04          /* for timeout use */
```

intel®

APPENDIX G
master FILE

This appendix lists an example of the **master** file, which you must edit to specify the configuration of your XENIX 286 system. Note that the **master** file that you receive with your XENIX system may be different. Chapter 6, "Adding Drivers to the Configuration", contains more information about the **master** file.

```
* The following devices are those that can be specified in the system
* description file.  The name specified must agree with the name shown.
*
* The first twelve entries in both the "bdevsw" and the "cdevsw" are
* reserved for use as block devices.  The last four of these entries
* are reserved for additional Intel devices and customer block devices.
* All block devices have the same "bdevsw" and "cdevsw" number.
* The "cmaj" number 1 is reserved for use by the memory driver.
*
*name     vsiz   msk     typ     hndlr   na   bmaj   cmaj   #   na   vec1   vec2   vec3   vec4
*  1       2      3       4        5      6     7      8     9   10    11     12     13     14
*-------------------------------------------------------------------------------------------
i215       1     0137    014      i215    0     0      0     2   -1   0005    0      0      0a
i216       1     0137    014      i216    0     2      2     2   -1   0005    0      0      0a
i214       1     0137    014      i214    0     3      3     2   -1   0005    0      0      0a
i208       1     0137    014      i208    0     4      4     2   -1   0003    0      0      0a
ramd       0     0136    054      ramd    0     5      5     1   -1      0    0      0      0a
xlog       1     0137    014      xlog    0     6      6     2   -1   0005    0      0      0a
*
* The next twelve entries in the "cdevsw" are reserved for character
* devices.  The "cmaj" number 12 is reserved for use by the tty driver.
*
lp         1     0132    004      lp      0     0     13     1   -1   0107    0      0      0a
i74        1     0137    004      i74     0     0     14     1   -1   0006    0      0      0a
i188       1     0137    004      i188    0     0     15     2   -1   0003   0002    0      0a
i552       1     0137    004      i552    0     0     16     1   -1   0004    0      0      0a
i278       1     0137    004      i278    0     0     17     1   -1   0003    0      0      0a
i544       1     0137    004      i544    0     0     18     4   -1   0003    0      0      0a
i534       1     0137    004      i534    0     0     19     4   -1   0003    0      0      0a
*
* These are Intel devices that use an interrupt vector but do not
* have any "bdevsw" or "cdevsw" entry.
*
debug      1       0       0      dbg     0     0      0     1   -1   0001    0      0      0a
slave7     1       0       0      sl      0     0      0     1   -1   0007    0      0      0a
*
* The following devices must not be specified in the system description
* file (xenixconf).  These are pseudo drivers and the clock driver.
*
memory     0      06     0324      mm      0    -1      1     1   0      0     0      0      0a
tty        0      027    0324      sy      0    -1     12     1   0      0     0      0      0a
clock      1      000    0321              0    -1     -1     1   0      0     0      0      0a
```

```
$$$
*
* The following are the line discipline table entries.
*
tty             ttopen  ttclose  ttread  ttwrite  ttioctl  ttin  ttout  nulldev

$$$$$
*
* The following entries form the alias table.
*
i215            disk
i188            serial
sm              sim
$$$
*
* The following entries form the tunable parameter table.
*
buffers         NBUF    0
sabufs          NSABUF    20
hashbuf         NHBUF    128
inodes          NINODE    100
files           NFILE    100
mounts          NMOUNT    6
coremap         CMAPSIZ    (NPROC*2)
swapmap         SMAPSIZ    (NPROC*2)
calls           NCALL    25
procs           NPROC    50
texts           NTEXT    40
clists          NCLIST    120
locks           NFLOCKS    50
maxproc         MAXUPRC 1    5
timezone        TIMEZONE    (8*60)
pages           NCOREL    0
daylight        DSTFLAG    1
cmask           CMASK    0
maxprocmem      MAXMEM    0
shdata          NSDSEGS    25
maxbuf          MAXBUF    192
```

intel®

APPENDIX H
xenixconf FILE

This appendix lists an example of the **xenixconf** file, which you edit to specify the configuration of your XENIX 286 system. Note that the **xenixconf** file that you receive with your XENIX system may be different. Chapter 6, "Adding Drivers to the Configuration", contains more information about the **xenixconf** file.

```
* THIS FILE CONTAINS CODE WHICH IS SPECIFIC TO THE
* INTEL 286/310 COMPUTER AND MAY REQUIRE MODIFICATION
* WHEN ADAPTING XENIX TO NEW HARDWARE.
*
*
*           Devices
*
i215        1
i534        0
i544        0
i188        0
i74         1
lp          1
ramd        1
debug       1
root        i215 1
pipe        i215 1
swap        i215 2 1 4104
*
*           Local parameters
*
timezone     (8*60)
daylight      1
cmask       0
```

intel®

APPENDIX I
c.c FILE

This appendix lists an example of the **c.c** source file, which specifies your XENIX 286 system configuration, and which is derived from **master** and **xenixconf** by running the program **config.** Note that the **c.c** file on your XENIX system may be different. More information about **c.c** is contained in Chapter 6, "Adding Drivers to the Configuration."

```
/*
 * Configuration information
 */


#define     NBUF        0
#define     NSABUF      20
#define     NHBUF       128
#define     NINODE      100
#define     NFILE       100
#define     NMOUNT      6
#define     CMAPSIZ     (NPROC*2)
#define     SMAPSIZ     (NPROC*2)
#define     NCALL       25
#define     NPROC       50
#define     NTEXT       40
#define     NCLIST      120
#define     NFLOCKS     50
#define     MAXUPRC     15
#define     TIMEZONE                (8*60)
#define     NCOREL      0
#define     DSTFLAG     1
#define     CMASK       0
#define     MAXMEM      0
#define     NSDSEGS     25
#define     MAXBUF      192

#include    "../h/param.h"
#include    "../h/conf.h"
#include    "../h/iobuf.h"

extern      nodev(),    nulldev(),    novec();


int         clock();
int         dbgintr();
int         i215intr();
int         i74intr();
int         lpintr();
```

```
int         (*vecintsw[])() =
{
            clock,
            dbgintr,
            novec,
            novec,
            novec,
            i215intr,
            i74intr,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
            novec,
```

```
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        novec,
        lpintr,
        novec,
};

extern  struct iobuf i215tab;
extern  i215open(),  i215close(),  i215init(),  i215read(),  i215write(),
        i215ioctl(),  i215strategy();
extern  i74open(),  i74close(),  i74init(),  i74read(),  i74write(),  i74ioctl();
extern  lpopen(),  lpclose(),  lpinit(),  lpwrite();
extern  struct iobuf ramdtab;
extern  ramdopen(),  ramdclose(),  ramdinit(),  ramdread(),  ramdwrite(),
        ramdstrategy();
extern  mmread(),  mmwrite();
extern  syopen(),  syread(),  sywrite(),  syioctl();

struct      bdevsw bdevsw[] =
{
/* 0*/      i215open,   i215close,   i215strategy,   &i215tab,
/* 1*/      nodev,      nodev,       nodev,           0,
/* 2*/      nodev,      nodev,       nodev,           0,
/* 3*/      nodev,      nodev,       nodev,           0,
/* 4*/      nodev,      nodev,       nodev,           0,
/* 5*/      ramdopen,   ramdclose,   ramdstrategy,    &ramdtab,
};
```

```
struct cdevsw cdevsw[] =
{
/* 0*/      i215open,   i215close,  i215read,   i215write,  i215ioctl,
/* 1*/      nulldev,    nulldev,    mmread,     mmwrite,    nodev,
/* 2*/      nodev,      nodev,      nodev,      nodev,      nodev,
/* 3*/      nodev,      nodev,      nodev,      nodev,      nodev,
/* 4*/      nodev,      nodev,      nodev,      nodev,      nodev,
/* 5*/      ramdopen,   ramdclose,  ramdread,   ramdwrite,  nodev,
/* 6*/      nodev,      nodev,      nodev,      nodev,      nodev,
/* 7*/      nodev,      nodev,      nodev,      nodev,      nodev,
/* 8*/      nodev,      nodev,      nodev,      nodev,      nodev,
/* 9*/      nodev,      nodev,      nodev,      nodev,      nodev,
/*10*/      nodev,      nodev,      nodev,      nodev,      nodev,
/*11*/      nodev,      nodev,      nodev,      nodev,      nodev,
/*12*/      syopen,     nulldev,    syread,     sywrite,    syioctl,
/*13*/      lpopen,     lpclose,    nodev,      lpwrite,    nodev,
/*14*/      i74open,    i74close,   i74read,    i74write,   i74ioctl,
};

int         bdevcnt =       6;
int         cdevcnt =       15;

dev_t       rootdev =       makedev(0,1);
dev_t       pipedev =       makedev(0,1);
dev_t       swapdev =       makedev(0,2);
daddr_t     swplo =  1;
int         nswap =  4104;


int         (*dinitsw[])() =
{
            i215init,
            i74init,
            lpinit,
            ramdinit,
            (int (*)())0
};

int         ttopen(), ttclose(), ttread(), ttwrite(), ttioctl(), ttin(),
            ttout();

struct      linesw linesw[] =
{
/*0*/       ttopen, ttclose, ttread, ttwrite, ttioctl, ttin, ttout, nulldev,
            0
};

int         linecnt = 1;

#include            "../h/space.h"
```

intel®

APPENDIX J
RELATED PUBLICATIONS

Copies of the following publications can be ordered from

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Guide to Using the iSBC 286/10 Single Board Computer, Order Number 146271 -- board options for interrupt and I/O configuration.

Overview of the XENIX 286 Operating System, Order Number 174385 -- XENIX history, XENIX uses, basic XENIX concepts, and an overview of other XENIX manuals.

XENIX 286 Installation and Configuration Guide, Order Number 174386 -- how to install XENIX on your hardware and tailor the XENIX configuration to your needs.

XENIX 286 User's Guide, Order Number 174387 -- a tutorial on the most-used parts of XENIX, including terminal conventions, the file system, the screen editor, and the shell.

XENIX 286 Visual Shell User's Guide, Order Number 174388 -- a XENIX command interface ("shell") that replaces the standard command syntax with a menu-driven command interpreter.

XENIX 286 System Administrator's Guide, Order Number 174389 -- how to perform system administrator tasks such as adding and removing users, backing up file systems, and troubleshooting system problems.

XENIX 286 Communications Guide, Order Number 174461 -- installing, using, and administering XENIX networking software.

XENIX 286 Reference Manual, Order Number 174390 -- all commands in the XENIX 286 Basic System.

XENIX 286 Programmer's Guide, Order Number 174391 -- XENIX 286 Extended System commands used for developing and maintaining programs.

XENIX 286 C Library Guide, Order Number 174542 -- standard subroutines used in programming with XENIX 286, including all system calls.

XENIX 286 Device Driver Guide, Order Number 174393 -- (this manual) how to write device drivers for XENIX 286 and add them to your system.

XENIX 286 Text Formatting Guide, Order Number 174541 -- XENIX 286 Extended System commands used for text processing and formatting.

C is described in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. One copy is supplied with Intel's XENIX product. Additional copies can be ordered from the publisher, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

intel®

INDEX

bdevsw, 2-11, I-3
buf structure, 5-2, E-1

cblock, 3-1
cdevsw, 2-11, I-4
clist, 3-1
copyin, 2-14
copyout, 2-14

device number, 2-10
dinitsw, 2-11

getc, 3-2

in, 2-13
inb, 2-13
interrupt handling, 2-7
iobuf structure, 5-5, F-1
iSBC 534 driver, 4-12
ixxxclose, 2-11, 3-5, 4-8, 5-13
ixxxinit, 2-11, 3-6, 4-7, 5-11
ixxxintr, 2-11, 3-9, 4-9, 5-16
ixxxioctl, 2-12, 4-11, 5-18
ixxxopen, 2-11, 3-5, 4-8, 5-12
ixxxparam, 4-7
ixxxproc, 4-10
ixxxread, 2-11, 3-6, 4-9, 5-17
ixxxstart, 3-8, 4-11, 5-15
ixxxstrategy, 2-12, 5-14
ixxxwrite, 2-12, 3-7, 4-9, 5-17

line discipline routines, 4-2

major macro, 2-10
makefiles, 6-7
master, 6-2, G-1
memory-mapped I/O, A-1
minor macro, 2-10

out, 2-13
outb, 2-13

peek routines, A-2
poke routines, A-3
proc table entry, 2-5
putc, 3-2
sleep, 2-6
spl routines, 2-8

ttclose, 4-4
ttin, 4-5
ttinit, 4-3
ttiocom, 4-5
ttioctl, 4-5
ttopen, 4-4
ttout, 4-6
ttread, 4-4
ttwrite, 4-4
tty structure, 4-2, B-1, C-1

u structure, 2-5, 2-14

vecintsw, 2-11, I-2

wakeup, 2-6

xenixconf, 6-6, H-1

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does this publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply ☐

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Parkway
Hillsboro, Oregon 97123

ISO-N TECHNICAL PUBLICATIONS HF2-1-830

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.

SOFTWARE

0471/7K/0685/WCP/AD